Available online at www.sciencedirect.com

ScienceDirect

Information and Computation 206 (2008) 620–651

Information and Computation

www.elsevier.com/locate/ic

Generalizing the Paige–Tarjan algorithm by abstract interpretation

Francesco Ranzato *, Francesco Tapparo

Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy

Received 27 December 2006; revised 11 August 2007
Available online 7 February 2008

Abstract

The Paige and Tarjan algorithm (PT) for computing the coarsest refinement of a state partition which is a bisimulation on some Kripke structure is well known. It is also well known in model checking that bisimulation is equivalent to strong preservation of CTL or, equivalently, of Hennessy–Milner logic. Drawing on these observations, we analyze the basic steps of the PT algorithm from an abstract interpretation perspective, which allows us to reason on strong preservation in the context of arbitrary (temporal) languages and of generic abstract models, possibly different from standard state partitions, specified by abstract interpretation. This leads us to design a generalized Paige–Tarjan algorithm, called GPT, for computing the minimal refinement of an abstract interpretation-based model that strongly preserves some given language. It turns out that PT is a straight instance of GPT on the domain of state partitions for the case of strong preservation of Hennessy–Milner logic. We provide a number of examples showing that GPT is of general use. We first show how a well-known efficient algorithm for computing stuttering equivalence can be viewed as a simple instance of GPT. We then instantiate GPT in order to design a new efficient algorithm for computing simulation equivalence that is competitive with the best available algorithms. Finally, we show how GPT allows to deal with strong preservation of new languages by providing an efficient algorithm that computes the coarsest refinement of a given partition that strongly preserves a language generated by the reachability operator.

1. Introduction

1.1. Motivations

The Paige and Tarjan [26] algorithm—in the paper denoted by PT—for efficiently computing the coarsest refinement of a given partition which is *stable* for a given state transition relation is well known. Its importance stems from the fact that PT actually computes *bisimulation equivalence*, because a partition P of a state space Σ is stable for a transition relation $\rightarrow$ on Σ if and only if P is a bisimulation equivalence on the transition system $\langle \Sigma, \rightarrow \rangle$. In particular, PT is used in model checking for reducing the state space of a Kripke structure

* Corresponding author.
E-mail addresses: ranzato@math.unipd.it (F. Ranzato), tapparo@math.unipd.it (F. Tapparo).

0890-5401/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.01.001

$\mathcal{K}$ because the quotient of $\mathcal{K}$ with respect to bisimulation equivalence *strongly preserves* temporal languages like CTL*, CTL and the whole μ-calculus [2,4]. This means that logical specifications can be checked on the abstract quotient model of $\mathcal{K}$ with no loss of precision. Paige and Tarjan first present the basic $O(|\rightarrow||\Sigma|)$-time PT algorithm. Then, they exploit a computational logarithmic improvement—inspired by Hopcroft's "process the smaller half" strategy [22] to minimize the number of states of deterministic finite automata—in order to design a $O(|\rightarrow|\log(|\Sigma|))$-time algorithm, which is usually referred to as Paige–Tarjan algorithm. It is important to remark that the logarithmic Paige–Tarjan algorithm is derived as an algorithmic refinement of PT that does not affect the correctness of the procedure which is instead proved for the basic PT algorithm. As shown in [28], it turns out that state partitions can be viewed as *domains in abstract interpretation* and strong preservation can be cast as *completeness in abstract interpretation*. Thus, our first aim was to make use of an "abstract interpretation eye" to understand why PT is a correct procedure for computing strongly preserving partitions.

1.2. The PT algorithm

Let us recall how PT works. Let $\mathrm{pre}_{\rightarrow}(X) = \{s \in \Sigma \mid \exists x \in X.\, s \rightarrow x\}$ denote the usual predecessor transformer on $\wp(\Sigma)$. A partition $P \in \mathrm{Part}(\Sigma)$ is PT stable when for any block $B \in P$, if $B' \in P$ then either $B \subseteq \mathrm{pre}_{\rightarrow}(B')$ or $B \cap \mathrm{pre}_{\rightarrow}(B') = \varnothing$. For a given subset $S \subseteq \Sigma$, $\mathrm{PTsplit}(S,P)$ denotes the partition obtained from P by replacing each block $B \in P$ with the blocks $B \cap \mathrm{pre}_{\rightarrow}(S)$ and $B \smallsetminus \mathrm{pre}_{\rightarrow}(S)$, where we also allow no actual splitting, that is, $\mathrm{PTsplit}(S,P) = P$. When $P \neq \mathrm{PTsplit}(S,P)$ the subset S is called a *splitter* for P. $\mathrm{Splitters}(P)$ denotes the set of splitters of P, while $\mathrm{PTrefiners}(P) \stackrel{\mathrm{def}}{=} \{S \in \mathrm{Splitters}(P) \mid \exists \{B_i\} \subseteq P.\, S = \cup_i B_i\}$. Then, the PT algorithm goes as follows.

ALGORITHM: PT

Input: partition $P \in \mathrm{Part}(\Sigma)$
while (P is not PT-stable) **do**
  choose $S \in \mathrm{PTrefiners}(P)$;
  $P := \mathrm{PTsplit}(S,P)$;
Output: P

The time complexity of PT is $O(|\rightarrow||\Sigma|)$ because the number of iterations of the while loop is bounded by $|\Sigma|$ and, by storing $\mathrm{pre}_{\rightarrow}(\{s\})$ for each $s \in \Sigma$, finding a PT refiner and performing a splitting step takes $O(|\rightarrow|)$ time.

1.3. An abstract interpretation perspective of PT

This work originated from a number of observations on the above PT algorithm. First, we may view the output $\mathrm{PT}(P)$ as the coarsest refinement of a partition P that strongly preserves CTL. For partitions of the state space Σ, namely standard abstract models in model checking, it is known that strong preservation of CTL is equivalent to strong preservation of (finitary) Hennessy–Milner logic HML [19], i.e., the language:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EX}\varphi.$$

The interpretation of HML is standard: p ranges over atomic propositions in AP, the semantic interpretation of the existential next operator EX is the predecessor transformer $\mathrm{pre}_{\rightarrow} : \wp(\Sigma) \rightarrow \wp(\Sigma)$. The initial partition P is induced by the interpretation of atoms, namely two states are in the same block of P iff they satisfy the same atoms. We observe that $\mathrm{PT}(P)$ indeed computes the coarsest partition P_{HML} that refines P and strongly preserves HML. Moreover, the partition P_{HML} corresponds to the state equivalence $\equiv_{\mathrm{HML}}$ induced by the semantics of HML: $s \equiv_{\mathrm{HML}} s'$ iff $\forall \varphi \in \mathrm{HML}.\, s \in [\![\varphi]\!] \Leftrightarrow s' \in [\![\varphi]\!]$. We also observe that P_{HML} is an abstraction belonging to the domain $\mathrm{Part}(\Sigma)$ of partitions of Σ of the standard state semantics of HML. Thus, our starting point was that PT can be viewed as an algorithm for computing the most abstract object of a particular abstract domain,

i.e., $\mathrm{Part}(\Sigma)$, that strongly preserves a particular language, i.e., HML. We make this view precise within Cousot and Cousot's abstract interpretation framework [5,6].

Previous work [28] introduced an abstract interpretation-based framework for reasoning on strong preservation of abstract models with respect to generic inductively defined languages. We showed that the lattice $\mathrm{Part}(\Sigma)$ of partitions of the state space Σ can be viewed as an abstraction, through some abstraction and concretization maps α and γ, of the lattice $\mathrm{Abs}(\wp(\Sigma))$ of abstract interpretations of $\wp(\Sigma)$. Thus, a partition $P \in \mathrm{Part}(\Sigma)$ is here viewed as a particular abstract domain $\gamma(P) \in \mathrm{Abs}(\wp(\Sigma))$. This leads to a precise correspondence between *forward complete* abstract interpretations and strongly preserving abstract models. Let us recall that completeness in abstract interpretation [5,6,14] encodes an ideal situation where no loss of precision occurs by approximating concrete computations on abstract domains. The problem of minimally refining an abstract model in order to get strong preservation of some language $\mathcal{L}$ can be cast as the problem of making an abstract interpretation $\mathcal{A}$ forward complete for the semantic operators of $\mathcal{L}$ through a minimal refinement of the abstract domain $\mathcal{A}$. It turns out that this latter completeness problem always admits a fixpoint solution. Hence, in our abstract interpretation framework, it turns out that for any $P \in \mathrm{Part}(\Sigma)$, the output $\mathrm{PT}(P)$ is the partition abstraction in $\mathrm{Part}(\Sigma)$ through α of the minimal refinement of the abstract domain $\gamma(P) \in \mathrm{Abs}(\wp(\Sigma))$ that is complete for the set Op_{HML} of semantic operators of the language HML, that is $Op_{\mathrm{HML}} = \{\cap, \complement, \mathrm{pre}_{\rightarrow}\}$ consists of intersection, complementation and precedessor operators. In particular, a partition P is PT stable iff the abstract domain $\gamma(P)$ is complete for the operators in Op_{HML}. Also, the following observation is crucial in our approach. The splitting operation $\mathrm{PTsplit}(S,P)$ can be viewed as the *best correct approximation* on $\mathrm{Part}(\Sigma)$ of a refinement operation $\mathrm{refine}_{op}(S,\cdot)$ of abstract domains: given a semantic operator op, the operation $\mathrm{refine}_{op}(S,A)$ refines an abstract domain A through a "op-refiner" $S \in A$ to the most abstract domain that contains both A and the image $op(S)$. In particular, P results to be PT stable iff the abstract domain $\gamma(P)$ cannot be refined with respect to the predecessor operator $\mathrm{pre}_{\rightarrow}$. Thus, if $\mathrm{refine}^{\mathrm{Part}}_{op}$ denotes the best correct approximation in $\mathrm{Part}(\Sigma)$ of refine_{op} then the PT algorithm can be reformulated as follows.

Input: partition $P \in \mathrm{Part}(\Sigma)$
while (the set of $\mathrm{pre}_{\rightarrow}$-refiners of $P \neq \varnothing$) **do**
  choose some $\mathrm{pre}_{\rightarrow}$-refiner $S \in \gamma(P)$;
  $P := \mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}_{\rightarrow}}(S,P)$;
Output: P

1.4. Main results

This abstract interpretation-based view of the PT algorithm leads us to generalize PT to:

(1) a generic domain $\mathcal{A}$ of abstract models that generalizes the role played in PT by the domain of state partitions $\mathrm{Part}(\Sigma)$;
(2) a generic set Op of operators on $\wp(\Sigma)$ that provides the semantics of some language $\mathcal{L}_{Op}$ and generalizes the role played in PT by the set Op_{HML} of operators of HML.

We design a generalized Paige–Tarjan refinement algorithm, called GPT, that, for any input abstract model $A \in \mathcal{A}$, computes the most abstract refinement of A in $\mathcal{A}$ which is strongly preserving for the language $\mathcal{L}_{Op}$. The correctness of GPT is guaranteed by some completeness conditions on $\mathcal{A}$ and Op. We provide a number of applications showing that GPT is an algorithmic scheme of general use.

We first show how GPT can be instantiated in order to get the well-known Groote–Vaandrager algorithm [17] that computes divergence blind stuttering equivalence in Kripke structures in $O(|{\rightarrow}||\Sigma|)$-time (this is the best known time bound). Divergence blind stuttering equivalence corresponds to branching bisimulation equivalence in process algebras that preserves the branching structure of processes by taking into account invisible events

[2,8,16]. It turns out that the Groote–Vaandrager algorithm corresponds to an instance of GPT where the abstract domain $\mathcal{A}$ is the lattice of partitions $\mathrm{Part}(\Sigma)$ and the set of operators is $Op = \{\cap, \complement, \mathbf{EU}\}$, where $\mathbf{EU}$ denotes the standard semantic interpretation of the existential until.

We then show how GPT allows to design a new simple and efficient algorithm for computing simulation equivalence in a Kripke structure. This algorithm is obtained as a consequence of the fact that simulation equivalence corresponds to strong preservation of the language:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EX}\varphi.$$

Therefore, in this instance of GPT the set of operators is $Op = \{\cap, \mathrm{pre}_{\rightarrow}\}$ and the abstract domain $\mathcal{A}$ is the lattice of disjunctive (i.e., precise for least upper bounds [6]) abstract domains of $\wp(\Sigma)$. It turns out that this algorithm can be implemented with time and space complexities that are comparable with those of the best available algorithms for computing simulation equivalence.

Finally, we demonstrate how GPT can solve novel strong preservation problems by considering strong preservation with respect to the language inductively generated by propositional logic and the reachability operator $\mathbf{EF}$. Here, we obtain a partition refinement algorithm, namely the abstract domain $\mathcal{A}$ is the lattice of partitions $\mathrm{Part}(\Sigma)$, while the set of operators is $Op = \{\cap, \complement, \mathbf{EF}\}$. We describe an implementation for this instance of GPT that leads to a $O(|\rightarrow||\Sigma|)$-time algorithm. This instance of GPT is also experimentally evaluated.

This paper is an extended and revised version of [27].

2. Background

2.1. Notation and preliminaries

Notations. Let X be any set. $\mathrm{Fun}(X)$ denotes the set of functions $f : X^n \to X$, for any $n = \sharp(f) \geqslant 0$, called arity of f. Following a standard convention, when $n = 0$, f is meant to be a specific object of X. If $f : X \to Y$ then the image of f is also denoted by $\mathrm{img}(f) = \{f(x) \in Y \mid x \in X\}$. When writing a set S of subsets of a given set, like a partition, S is often written in a compact form like $\{1, 12, 13\}$ or $\{[1], [12], [13]\}$ that stands for $\{\{1\}, \{1, 2\}, \{1, 3\}\}$. The complement operator for the universe set X is $\complement : \wp(X) \to \wp(X)$, where $\complement(S) = X \smallsetminus S$.

Orders. Let $\langle P, \leqslant\rangle$ be a poset. Posets are often denoted by $P_{\leqslant}$. We use the symbol $(\sqsubset)\ \sqsubseteq$ to denote (strict) pointwise ordering between functions: if X is any set and $f, g : X \to P$ then $f \sqsubseteq g$ if for all $x \in X$, $f(x) \leqslant g(x)$. A mapping $f : P \to Q$ on posets is continuous when f preserves least upper bounds (lub's) of countable chains in P, while, dually, it is co-continuous when f preserves greatest lower bounds (glb's) of countable chains in P. A complete lattice $C_{\leqslant}$ is also denoted by $\langle C, \leqslant, \vee, \wedge, \top, \bot\rangle$ where $\vee$, $\wedge$, $\top$ and $\bot$ denote, respectively, lub, glb, greatest element and least element in C. A function $f : C \to D$ between complete lattices is additive (co-additive) when f preserves least upper (greatest lower) bounds. We denote by $\mathrm{lfp}(f)$ and $\mathrm{gfp}(f)$, respectively, the least and greatest fixpoint, when they exist, of an operator f on a poset.

Partitions. A partition P of a set Σ is a set of nonempty subsets of Σ, called blocks, that are pairwise disjoint and whose union gives Σ. $\mathrm{Part}(\Sigma)$ denotes the set of partitions of Σ. $\mathrm{Part}(\Sigma)$ is endowed with the following standard partial order $\preceq$: $P_1 \preceq P_2$, i.e., P_2 is coarser than P_1 (or P_1 refines P_2) iff $\forall B \in P_1.\exists B' \in P_2.\ B \subseteq B'$. It is well known that $\langle \mathrm{Part}(\Sigma), \preceq, \curlywedge, \curlyvee, \{\Sigma\}, \{\{s\}\}_{s\in\Sigma}\rangle$ is a complete lattice, where $P_1 \curlywedge P_2 = \{B_1 \cap B_2 \mid B_1 \in P_1,\ B_2 \in P_2,\ B_1 \cap B_2 \neq \varnothing\}$.

Kripke structures. A transition system $\mathcal{T} = (\Sigma, \rightarrow)$ consists of a (possibly infinite) set Σ of states and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. As usual [4], we assume that the relation $\rightarrow$ is total, i.e., for any $s \in \Sigma$ there exists some $t \in \Sigma$ such that $s \rightarrow t$, so that any maximal path in $\mathcal{T}$ is necessarily infinite. The pre/post transformers on $\wp(\Sigma)$ are defined as usual:

- $\mathrm{pre}_{\rightarrow} \stackrel{\mathrm{def}}{=} \lambda Y.\{a \in \Sigma \mid \exists b \in Y.\ a \rightarrow b\}$,
- $\widetilde{\mathrm{pre}}_{\rightarrow} \stackrel{\mathrm{def}}{=} \complement \circ \mathrm{pre}_{\rightarrow} \circ \complement = \lambda Y.\{a \in \Sigma \mid \forall b \in \Sigma.(a \rightarrow b \Rightarrow b \in Y)\}$,

– $\mathrm{post}_{\rightarrow} \stackrel{\mathrm{def}}{=} \lambda Y.\{b \in \Sigma \mid \exists a \in Y.\ a \rightarrow b\}$,

– $\widetilde{\mathrm{post}}_{\rightarrow} \stackrel{\mathrm{def}}{=} \complement \circ \mathrm{post}_{\rightarrow} \circ \complement = \lambda Y.\{b \in \Sigma \mid \forall a \in \Sigma.(a \rightarrow b \Rightarrow a \in Y)\}$.

Let us remark that $\mathrm{pre}_{\rightarrow}$ and $\mathrm{post}_{\rightarrow}$ are additive operators on $\wp(\Sigma)_{\subseteq}$ while $\widetilde{\mathrm{pre}}_{\rightarrow}$ and $\widetilde{\mathrm{post}}_{\rightarrow}$ are co-additive. When clear from the context, subscripts in pre/post transformers are sometimes omitted.

Given a set AP of atomic propositions (of some language), a Kripke structure $\mathcal{K} = (\Sigma, \rightarrow, \ell)$ over AP consists of a transition system $(\Sigma, \rightarrow)$ together with a state labeling function $\ell : \Sigma \rightarrow \wp(AP)$. We use the following notation: for any $s \in \Sigma$, $[s]_\ell \stackrel{\mathrm{def}}{=} \{s' \in \Sigma \mid \ell(s) = \ell(s')\}$, while $P_\ell \stackrel{\mathrm{def}}{=} \{[s]_\ell \mid s \in \Sigma\} \in \mathrm{Part}(\Sigma)$ denotes the state partition that is induced by ℓ.

The notation $s \models^{\mathcal{K}} \varphi$ means that a state $s \in \Sigma$ satisfies in $\mathcal{K}$ a state formula φ of some language $\mathcal{L}$, where the specific definition of the satisfaction relation $\models^{\mathcal{K}}$ depends on the language $\mathcal{L}$ (interpretations of standard logical/temporal operators like next, until, finally, etc. can be found in [4]).

2.2. Abstract interpretation and completeness

2.2.1. Abstract domains

In standard Cousot and Cousot's abstract interpretation, abstract domains can be equivalently specified either by Galois connections, i.e., adjunctions, or by upper closure operators (uco's) [5,6]. Let us recall these standard notions.

Galois connections and insertions. If A and C are posets and $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ are monotone functions such that $\forall c \in C.\ c \leqslant_C \gamma(\alpha(c))$ and $\alpha(\gamma(a)) \leqslant_A a$ then the quadruple (α, C, A, γ) is called a Galois connection (GC for short) between C and A. If in addition $\alpha \circ \gamma = \lambda x.x$ then (α, C, A, γ) is a Galois insertion (GI for short) of A in C. In a GI, γ is 1–1 and α is onto. Let us also recall that the notion of GC is equivalent to that of adjunction: if $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ then (α, C, A, γ) is a GC iff $\forall c \in C. \forall a \in A.\ \alpha(c) \leqslant_A a \Leftrightarrow c \leqslant_C \gamma(a)$. The map α (γ) is called the left- (right-) adjoint to γ (α). It turns out that one adjoint map α/γ uniquely determines the other adjoint map γ/α as follows. On the one hand, a map $\alpha : C \rightarrow A$ admits a necessarily unique right-adjoint map $\gamma : A \rightarrow C$ iff α preserves arbitrary lub's; in this case, we have that $\gamma \stackrel{\mathrm{def}}{=} \lambda a.\ \vee_C \{c \in C \mid \alpha(c) \leqslant_A a\}$. On the other hand, a map $\gamma : A \rightarrow C$ admits a necessarily unique left-adjoint map $\alpha : C \rightarrow A$ iff γ preserves arbitrary glb's; in this case, $\alpha \stackrel{\mathrm{def}}{=} \lambda c.\ \wedge_A \{a \in A \mid c \leqslant_C \gamma(a)\}$. In particular, in any GC (α, C, A, γ) between complete lattices it turns out that α is additive and γ is co-additive.

We assume the standard abstract interpretation framework, where concrete and abstract domains, C and A, are complete lattices related by abstraction and concretization maps α and γ forming a GC (α, C, A, γ). A is called an abstraction of C and C a concretization of A. The ordering relations on concrete and abstract domains describe the relative precision of domain values: $x \leqslant y$ means that y is an approximation of x or, equivalently, x is more precise than y. Galois connections relate the concrete and abstract notions of relative precision: an abstract value $a \in A$ approximates a concrete value $c \in C$ when $\alpha(c) \leqslant_A a$, or, equivalently (by adjunction), $c \leqslant_C \gamma(a)$. As a key consequence of requiring a Galois connection, it turns out that $\alpha(c)$ is the best possible approximation in A of c, that is $\alpha(c) = \wedge\{a \in A \mid c \leqslant_C \gamma(a)\}$ holds. If (α, C, A, γ) is a GI then each value of the abstract domain A is useful in representing C, because all the values in A represent distinct members of C, being γ 1–1. Any GC can be lifted to a GI by identifying in an equivalence class those values of the abstract domain with the same concretization. $\mathrm{Abs}(C)$ denotes the set of abstract domains of C and we write $A \in \mathrm{Abs}(C)$ to mean that the abstract domain A is related to C through a GI (α, C, A, γ).

An abstract domain $A \in \mathrm{Abs}(C)$ is disjunctive when the corresponding concretization map γ is additive or, equivalently, when the image $\gamma(A) \subseteq C$ is closed under arbitrary lub's of C. We denote by $\mathrm{dAbs}(C)$ the subset of disjunctive abstract domains.

Closure operators. An (upper) closure operator, or simply a closure, on a poset $P_{\leqslant}$ is an operator $\mu : P \rightarrow P$ that is monotone, idempotent and extensive, i.e., $\forall x \in P.\ x \leqslant \mu(x)$. Dually, lower closure operators are monotone, idempotent, and restrictive, i.e., $\forall x \in P.\ \mu(x) \leqslant x$. $\mathrm{uco}(P)$ denotes the set of closure operators on P. Let $\langle C, \leqslant, \vee, \wedge, \top, \bot\rangle$ be a complete lattice. A closure $\mu \in \mathrm{uco}(C)$ is uniquely determined by its image $\mathrm{img}(\mu)$, which coincides with its set of fixpoints, as follows: $\mu = \lambda y.\ \wedge\{x \in \mathrm{img}(\mu) \mid y \leqslant x\}$. Also, $X \subseteq C$ is the image of some closure operator μ_X on C iff X is a Moore-family of C (or Moore-closed), i.e., $X = \mathcal{M}(X) \stackrel{\mathrm{def}}{=} \{\wedge S \mid S \subseteq X\}$— where $\wedge\varnothing = \top \in \mathcal{M}(X)$. In other terms, X is a Moore-family of C when X is meet-closed. In this case, $\mu_X =$

$\lambda y. \wedge \{x \in X \mid y \leqslant x\}$ is the corresponding closure operator on C. For any $X \subseteq C$, $\mathcal{M}(X)$ is called the Moore-closure of X in C, i.e., $\mathcal{M}(X)$ is the least (with respect to set inclusion) subset of C which contains X and is a Moore-family of C. Moreover, it turns out that for any $\mu \in \mathrm{uco}(C)$ and any Moore-family $X \subseteq C$, $\mu_{\mathrm{img}(\mu)} = \mu$ and $\mathrm{img}(\mu_X) = X$. Thus, closure operators on C are in bijection with Moore-families of C. This allows us to consider a closure operator $\mu \in \mathrm{uco}(C)$ both as a function $\mu : C \to C$ and as a Moore-family $\mathrm{img}(\mu) \subseteq C$. This is particularly useful and does not give rise to ambiguity since one can distinguish the use of a closure μ as function or set according to the context.

If C is a complete lattice then $\mathrm{uco}(C)$ endowed with the pointwise ordering $\sqsubseteq$ is a complete lattice denoted by $\langle \mathrm{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x\rangle$, where for every $\mu, \eta \in \mathrm{uco}(C)$, $\{\mu_i\}_{i\in I} \subseteq \mathrm{uco}(C)$ and $x \in C$:

- $\mu \sqsubseteq \eta$ iff $\forall y \in C.\ \mu(y) \leqslant \eta(y)$ iff $\mathrm{img}(\eta) \subseteq \mathrm{img}(\mu)$;
- $(\sqcap_{i\in I}\mu_i)(x) = \wedge_{i\in I}\mu_i(x)$ and $\mathrm{img}(\sqcap_{i\in I}\mu_i) = \mathcal{M}(\cup_{i\in I}\mathrm{img}(\mu_i))$;
- $\mathrm{img}(\sqcup_{i\in I}\mu_i) = \cap_{i\in I}\mathrm{img}(\mu_i)$;
- $\lambda x.\top$ is the greatest element, whereas $\lambda x.x$ is the least element.

A closure $\mu \in \mathrm{uco}(C)$ is disjunctive when μ preserves arbitrary lub's or, equivalently, when $\mathrm{img}(\mu)$ is join-closed, that is $\{\vee X \mid X \subseteq \mathrm{img}(\mu)\} = \mathrm{img}(\mu)$. Hence, a subset $X \subseteq C$ is the image of a disjunctive closure on C iff X is both meet- and join-closed. If C is completely distributive—this is the case, for example, of a lattice $\langle \wp(\Sigma), \subseteq\rangle$ for some set Σ—then the greatest (with respect to $\sqsubseteq$) disjunctive closure $\mathbb{D}(S)$ that contains a set $S \subseteq C$ is obtained by closing S under meets and joins, namely $\mathbb{D}(S) \stackrel{\mathrm{def}}{=} \{\vee X \mid X \subseteq \mathcal{M}(S)\}$ [6].

Closures are equivalent to Galois insertions. It is well known since [6] that abstract domains can be equivalently specified either as Galois insertions or as closures. These two approaches are completely equivalent. On the one hand, if $\mu \in \mathrm{uco}(C)$ and A is a complete lattice which is isomorphic to $\mathrm{img}(\mu)$, where $\iota : \mathrm{img}(\mu) \to A$ and $\iota^{-1} : A \to \mathrm{img}(\mu)$ provide the isomorphism, then $(\iota \circ \mu, C, A, \iota^{-1})$ is a GI. On the other hand, if (α, C, A, γ) is a GI then $\mu_A \stackrel{\mathrm{def}}{=} \gamma \circ \alpha \in \mathrm{uco}(C)$ is the closure associated with A such that $\langle \mathrm{img}(\mu_A), \leqslant_C\rangle$ is a complete lattice which is isomorphic to $\langle A, \leqslant_A\rangle$. Furthermore, these two constructions are inverse of each other. Let us also remark that an abstract domain A is disjunctive iff the uco μ_A is disjunctive. Given an abstract domain A specified by a GI (α, C, A, γ), its associated closure $\gamma \circ \alpha$ on C can be thought of as the "logical meaning" of A in C, since this is shared by any other abstract representation for the objects of A. Thus, the closure operator approach is particularly convenient when reasoning about properties of abstract domains independently from the representation of their objects.

The lattice of abstract domains. Abstract domains specified by GIs can be pre-ordered with respect to precision as follows: if $A_1, A_2 \in \mathrm{Abs}(C)$ then A_1 is more precise (or concrete) than A_2 (or A_2 is an abstraction of A_1) when $\mu_{A_1} \sqsubseteq \mu_{A_2}$. The pointwise ordering $\sqsubseteq$ between uco's corresponds therefore to the standard ordering used to compare abstract domains with respect to their precision. Also, A_1 and A_2 are equivalent, denoted by $A_1 \simeq A_2$, when their associated closures coincide, i.e., $\mu_{A_1} = \mu_{A_2}$. Hence, the quotient $\mathrm{Abs}(C)_{/\simeq}$ gives rise to a poset that, by a slight abuse of notation, is simply denoted by $\langle \mathrm{Abs}(C), \sqsubseteq\rangle$. Thus, when we write $A \in \mathrm{Abs}(C)$ we mean that A is any representative of an equivalence class in $\mathrm{Abs}(C)_{/\simeq}$ and is specified by a Galois insertion (α, C, A, γ). It turns out that $\langle \mathrm{Abs}(C), \sqsubseteq\rangle$ is a complete lattice, called the lattice of abstract domains of C [5,6], because it is isomorphic to the complete lattice $\langle \mathrm{uco}(C), \sqsubseteq\rangle$. Lub's and glb's in $\mathrm{Abs}(C)$ have therefore the following reading as operators on domains. Let $\{A_i\}_{i\in I} \subseteq \mathrm{Abs}(C)$: (i) $\sqcup_{i\in I}A_i$ is the most concrete among the domains which are abstractions of all the A_i's; (ii) $\sqcap_{i\in I}A_i$ is the most abstract among the domains which are more concrete than every A_i—this latter domain is also known as reduced product [6] of all the A_i's.

2.2.2. Completeness in abstract interpretation

Correct abstract interpretations. Let C be a concrete domain, $f : C \to C$ be a concrete semantic function[1] and $f^\sharp : A \to A$ be a corresponding abstract function on an abstract domain $A \in \mathrm{Abs}(C)$ specified by a GI

[1] For simplicity of notation we consider here unary functions since the extension to generic n-ary functions is straightforward.

(α, C, A, γ). Then, $\langle A, f^\sharp\rangle$ is a sound (or correct) abstract interpretation when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds. The abstract function $f^\sharp$ is called a correct approximation on A of f. This means that a concrete computation $f(c)$ can be correctly approximated in A by $f^\sharp(\alpha(c))$, namely $\alpha(f(c)) \leqslant_A f^\sharp(\alpha(c))$. An abstract function $f_1^\sharp : A \to A$ is more precise than $f_2^\sharp : A \to A$ when $f_1^\sharp \sqsubseteq f_2^\sharp$. Since $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds iff $\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$ holds, the abstract function $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \to A$ is called the best correct approximation of f in A.

Complete abstract interpretations. Completeness in abstract interpretation corresponds to requiring that, in addition to soundness, no loss of precision occurs when $f(c)$ is approximated in A by $f^\sharp(\alpha(c))$. Thus, completeness of $f^\sharp$ for f is encoded by the equation $\alpha \circ f = f^\sharp \circ \alpha$. This is also called backward completeness because a dual form of forward completeness may be considered. As a very simple example, let us consider the abstract domain *Sign* representing the sign of an integer variable, namely $Sign = \{\bot, \mathbb{Z}_{\leqslant 0}, 0, \mathbb{Z}_{\geqslant 0}, \top\} \in \mathrm{Abs}(\wp(\mathbb{Z})_\subseteq)$. Let us consider the binary concrete operation of integer addition on sets of integers, that is $X + Y \stackrel{\text{def}}{=} \{x + y \mid x \in X,\ y \in Y\}$, and the square operator on sets of integers, that is $X^2 \stackrel{\text{def}}{=} \{x^2 \mid x \in X\}$. It turns out that the best correct approximation $+^{Sign}$ of integer addition in *Sign* is sound but not complete—because $\alpha(\{-1\} + \{1\}) = 0 <_{Sign} \top = \alpha(\{-1\}) +^{Sign} \alpha(\{1\})$—while it is easy to check that the best correct approximation of the square operation in *Sign* is instead complete. Let us also recall that backward completeness implies fixpoint completeness, meaning that if $\alpha \circ f = f^\sharp \circ \alpha$ then $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$.

A dual form of completeness can be considered. The soundness condition $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ can be equivalently formulated as $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$. Forward completeness for $f^\sharp$ corresponds to requiring that the equation $f \circ \gamma = \gamma \circ f^\sharp$ holds, and therefore means that no loss of precision occurs when a concrete computation $f(\gamma(a))$, for some abstract value $a \in A$, is approximated in A by $f^\sharp(a)$. Let us notice that backward and forward completeness are orthogonal concepts. In fact: (1) we observed above that $+^{Sign}$ is not backward complete while it is forward complete because for any $a_1, a_2 \in Sign$, $\gamma(a_1) + \gamma(a_2) = \gamma(a_1 +^{Sign} a_2)$: for instance, $\gamma(\mathbb{Z}_{\geqslant 0}) + \gamma(\mathbb{Z}_{\geqslant 0}) = \mathbb{Z}_{\geqslant 0} = \gamma(\mathbb{Z}_{\geqslant 0} +^{Sign} \mathbb{Z}_{\geqslant 0})$; (2) the best correct approximation $(\cdot)^{2_{Sign}}$ of the square operator on *Sign* is not forward complete because $\gamma(\mathbb{Z}_{\geqslant 0})^2 \subsetneq \gamma(\mathbb{Z}_{\geqslant 0}) = \gamma((\mathbb{Z}_{\geqslant 0})^{2_{Sign}})$, while, as observed above, it is instead backward complete.

Completeness is an abstract domain property. Giacobazzi et al. [14] observed that completeness uniquely depends upon the abstraction map, i.e., upon the abstract domain. This means that if $f^\sharp$ is backward complete for f then the best correct approximation f^A of f in A is backward complete as well, and, in this case, $f^\sharp$ indeed coincides with f^A. Hence, for any abstract domain A, one can define a backward complete abstract operation $f^\sharp$ on A if and only if f^A is backward complete. An abstract domain $A \in \mathrm{Abs}(C)$ is therefore defined to be backward complete for f iff the equation $\alpha \circ f = f^A \circ \alpha$ holds. This simple observation makes backward completeness an abstract domain property, namely an intrinsic characteristic of the abstract domain. Let us observe that $\alpha \circ f = f^A \circ \alpha$ holds iff $\gamma \circ \alpha \circ f = \gamma \circ f^A \circ \alpha = \gamma \circ \alpha \circ f \circ \gamma \circ \alpha$ holds, so that A is backward complete for f when $\mu_A \circ f = \mu_A \circ f \circ \mu_A$. Thus, a closure $\mu \in \mathrm{uco}(C)$, that defines some abstract domain, is backward complete for f when $\mu \circ f = \mu \circ f \circ \mu$ holds. Analogous observations apply to forward completeness, which is also an abstract domain property: $A \in \mathrm{Abs}(C)$ is forward complete for f (or forward f-complete) when $f \circ \mu_A = \mu_A \circ f \circ \mu_A$, while a closure $\mu \in \mathrm{uco}(C)$ is forward complete for f when $f \circ \mu = \mu \circ f \circ \mu$ holds.

2.3. Shells

Refinements of abstract domains have been studied from the beginning of abstract interpretation [5,6] and led to the notion of shell of abstract domains [10,13,14]. Given a generic poset $P_\leqslant$ of semantic objects—where $x \leqslant y$ intuitively means that x is a "refinement" of y—and a property $\mathcal{P} \subseteq P$ of these objects, the generic notion of *shell* is as follows: the $\mathcal{P}$-shell of an object $x \in P$ is defined to be an object $s_x \in P$ such that:

(i) s_x satisfies the property $\mathcal{P}$,
(ii) s_x is a refinement of x, and
(iii) s_x is the greatest among the objects in P satisfying (i) and (ii).

Note that if a $\mathcal{P}$-shell exists then it is unique. Moreover, if the $\mathcal{P}$-shell exists for any object in P then it turns out that the operator that maps any $x \in P$ to its $\mathcal{P}$-shell is a lower closure operator on $\mathcal{P}$, being monotone, idempotent

and reductive: this is called the $\mathcal{P}$-*shell refinement* operator. We will be interested in shells of abstract domains and partitions, namely shells in the complete lattices of abstract domains and partitions. Given a state space Σ and a partition property $\mathcal{P} \subseteq \mathrm{Part}(\Sigma)$, the $\mathcal{P}$-shell of $P \in \mathrm{Part}(\Sigma)$ is the coarsest refinement of P satisfying $\mathcal{P}$, when this exists. Also, given a concrete domain C and a domain property $\mathcal{P} \subseteq \mathrm{Abs}(C)$, the $\mathcal{P}$-shell of $A \in \mathrm{Abs}(C)$, when this exists, is the most abstract domain that satisfies $\mathcal{P}$ and refines A. As an important example, Giacobazzi et al. [14] constructively showed that backward complete shells always exist when the concrete functions are continuous.

Disjunctive shells. Consider the abstract domain property of being disjunctive, namely $\mathrm{dAbs}(C) \subseteq \mathrm{Abs}(C)$. As already observed in [6], if C is a completely distributive lattice[2] then any abstract domain $A \in \mathrm{Abs}(C)$ can be refined to its disjunctive completion $\mathrm{dc}(A) \stackrel{\mathrm{def}}{=} \{\vee_C S \mid S \subseteq \gamma(A)\}$. This means that $\mathrm{dc}(A)$ is the most abstract domain that refines A and is disjunctive, namely it is the disjunctive shell of A. Hence, the disjunctive shell operator $\mathscr{S}_{\mathrm{dis}} : \mathrm{Abs}(C) \to \mathrm{Abs}(C)$ is defined as follows:

$$\mathscr{S}_{\mathrm{dis}}(A) \stackrel{\mathrm{def}}{=} \sqcup \{X \in \mathrm{Abs}(C) \mid X \sqsubseteq A,\ X \text{ is disjunctive}\}.$$

Forward complete shells. Let $F \subseteq \mathrm{Fun}(C)$ (thus functions in F may have any arity) and $S \in \wp(C)$. We denote by $F(S) \in \wp(C)$ the image of F on S, i.e., $F(S) \stackrel{\mathrm{def}}{=} \{f(\vec{s}) \mid f \in F,\ \vec{s} \in S^{\sharp(f)}\}$, and we say that S is F-closed when $F(S) \subseteq S$. An abstract domain $A \in \mathrm{Abs}(C)$ is forward F-complete when A is forward complete for any $f \in F$. Let us observe that F-completeness for an abstract domain A means that the image $\gamma(A)$ is closed under the image of functions in F, namely $F(\gamma(A)) \subseteq \gamma(A)$. Also note that when $k : C^0 \to C$, i.e., $k \in C$ is a constant, A is forward k-complete iff k is precisely represented in A, i.e., $\gamma(\alpha(k)) = k$. Let us finally note that any abstract domain is always forward meet-complete because any uco is Moore-closed.

The (forward) F-complete shell operator $\mathscr{S}_F : \mathrm{Abs}(C) \to \mathrm{Abs}(C)$ is defined as follows:

$$\mathscr{S}_F(A) \stackrel{\mathrm{def}}{=} \sqcup \{X \in \mathrm{Abs}(C) \mid X \sqsubseteq A,\ X \text{ is forward } F\text{-complete}\}.$$

As observed in [12,28], it turns out that for any abstract domain A, $\mathscr{S}_F(A)$ is forward F-complete, namely forward complete shells always exist. When C is finite, note that for the meet operator $\wedge : C^2 \to C$ we have that, for any F, $\mathscr{S}_F = \mathscr{S}_{F \cup \{\wedge\}}$, because uco's (that is, abstract domains) are meet-closed.

A forward complete shell $\mathscr{S}_F(A)$ is a more concrete abstraction than A. How to characterize $\mathscr{S}_F(A)$? As shown in [28], forward complete shells admit a constructive fixpoint characterization. Let $F^{\mathcal{M}} : \mathrm{Abs}(C) \to \mathrm{Abs}(C)$ be defined as follows: $F^{\mathcal{M}}(X) \stackrel{\mathrm{def}}{=} \mathcal{M}(F(\gamma(X)))$, namely $F^{\mathcal{M}}(X)$ is the most abstract domain that contains the image of F on $\gamma(X)$. Given $A \in \mathrm{Abs}(C)$, we consider the operator $F_A : \mathrm{Abs}(C) \to \mathrm{Abs}(C)$ defined by the reduced product $F_A(X) \stackrel{\mathrm{def}}{=} A \sqcap F^{\mathcal{M}}(X)$. Let us observe that $F_A(X) = \mathcal{M}(\gamma(A) \cup F(\gamma(X)))$ and that F_A is monotone and therefore admits the greatest fixpoint. This greatest fixpoint provides the forward F-complete shell of A:

$$\mathscr{S}_F(A) = \mathrm{gfp}(F_A). \tag{2.1}$$

Example 2.1. Let $\Sigma = \{1,2,3,4\}$ and $R \subseteq \Sigma \times \Sigma$ be the relation $\{(1,2),(2,3),(3,4),(4,4)\}$. Let us consider the post transformer $\mathrm{post}_R : \wp(\Sigma) \to \wp(\Sigma)$. Consider the abstract domain $A = \{\varnothing, 2, 1234\} \in \mathrm{Abs}(\wp(\Sigma)_{\subseteq})$. We have that $\mathscr{S}_{\mathrm{post}_R}(A) = \{\varnothing, 2, 3, 4, 34, 234, 1234\}$ because by (2.1):

$$\begin{aligned}
X_0 &= \{1234\} \quad \text{(most abstract domain)},\\
X_1 &= \mathcal{M}(A \cup \mathrm{post}_R(X_0)) = \mathcal{M}(A \cup \{234\}) = \{\varnothing, 2, 234, 1234\},\\
X_2 &= \mathcal{M}(A \cup \mathrm{post}_R(X_1)) = \mathcal{M}(A \cup \{\varnothing, 3, 34, 234\}) = \{\varnothing, 2, 3, 34, 234, 1234\},\\
X_3 &= \mathcal{M}(A \cup \mathrm{post}_R(X_2)) = \mathcal{M}(A \cup \{\varnothing, 3, 4, 34, 234\}) = \{\varnothing, 2, 3, 4, 34, 234, 1234\},\\
X_4 &= \mathcal{M}(A \cup \mathrm{post}_R(X_3)) = \mathcal{M}(A \cup \{\varnothing, 3, 4, 34, 234\}) = X_3 \quad \text{(greatest fixpoint)}.
\end{aligned}$$

[2] This means that in C arbitrary glb's distribute over arbitrary lub's—any powerset, ordered with respect to super-/sub-set relation, is completely distributive.

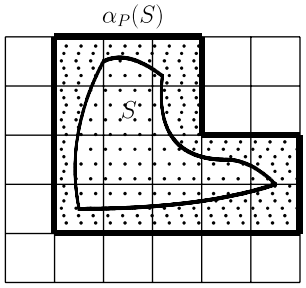

Fig. 1. Partitions as abstract domains.

3. Generalized strong preservation

Let us recall from [28] how partitions, i.e., standard abstract models, can be viewed as particular abstract domains and how strong preservation in standard abstract model checking can be cast as forward completeness of abstract interpretations.

3.1. Partitions as abstract domains

Let Σ be any (possibly infinite) set of system states. As shown in [28], it turns out that the lattice of state partitions $\mathrm{Part}(\Sigma)$ can be viewed as an abstraction of the lattice of abstract domains $\mathrm{Abs}(\wp(\Sigma))$. This is important for our goal of performing an *abstract fixpoint computation* on the abstract lattice $\mathrm{Part}(\Sigma)$ of a forward complete shell in $\mathrm{Abs}(\wp(\Sigma))$.

A partition $P \in \mathrm{Part}(\Sigma)$ can be viewed as an abstraction of $\wp(\Sigma)_{\subseteq}$ as follows: any $S \subseteq \Sigma$ is over-approximated by the unique minimal cover of S in P, namely by the union of all the blocks $B \in P$ such that $B \cap S \neq \varnothing$. A graphical example is depicted in Fig. 1. This abstraction is formalized by a GI $(\alpha_P, \wp(\Sigma)_{\subseteq}, \wp(P)_{\subseteq}, \gamma_P)$ where:

$$\alpha_P(S) \stackrel{\text{def}}{=} \{B \in P \mid B \cap S \neq \varnothing\}, \quad \gamma_P(\mathcal{B}) \stackrel{\text{def}}{=} \cup_{B \in \mathcal{B}} B.$$

We can therefore define a function $\mathrm{pad} : \mathrm{Part}(\Sigma) \to \mathrm{Abs}(\wp(\Sigma))$ that maps any partition P to an abstract domain $\mathrm{pad}(P)$ which is called *partitioning*. In general, an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$ is called partitioning when A is equivalent to an abstract domain $\mathrm{pad}(P)$ for some partition $P \in \mathrm{Part}(\Sigma)$. Accordingly, a closure $\mu \in \mathrm{uco}(\wp(\Sigma))$ that coincides with $\gamma_P \circ \alpha_P$, for some partition P, is called partitioning. It can be shown that an abstract domain A is partitioning iff its image $\gamma(A)$ is closed under complements, that is, $\forall S \in \gamma(A).\ \complement(S) \in \gamma(A)$. We denote by $\mathrm{Abs}^{\mathrm{par}}(\wp(\Sigma))$ and $\mathrm{uco}^{\mathrm{par}}(\wp(\Sigma))$ the sets of, respectively, partitioning abstract domains and closures on $\wp(\Sigma)$.

Partitions can thus be viewed as representations of particular abstract domains. On the other hand, it turns out that abstract domains can be abstracted to partitions. An abstract domain $A \in \mathrm{Abs}(\wp(\Sigma)_{\subseteq})$ induces a state equivalence $\equiv_A$ on Σ by identifying those states that cannot be distinguished by A:

$$s \equiv_A s' \quad \text{iff} \quad \alpha(\{s\}) = \alpha(\{s'\}).$$

For any $s \in \Sigma$, $[s]_A \stackrel{\text{def}}{=} \{s' \in \Sigma \mid \alpha(\{s\}) = \alpha(\{s'\})\}$ is a block of the state partition $\mathrm{par}(A)$ induced by A:

$$\mathrm{par}(A) \stackrel{\text{def}}{=} \{[s]_A \mid s \in \Sigma\}.$$

Thus, $\mathrm{par} : \mathrm{Abs}(\wp(\Sigma)) \to \mathrm{Part}(\Sigma)$ is a mapping from abstract domains to partitions.

Example 3.1. Let $\Sigma = \{1, 2, 3, 4\}$ and let us specify abstract domains as uco's on $\wp(\Sigma)$. The abstract domains $A_1 = \{\varnothing, 12, 3, 4, 1234\}$, $A_2 = \{\varnothing, 12, 3, 4, 34, 1234\}$, $A_3 = \{\varnothing, 12, 3, 4, 34, 123, 124, 1234\}$, $A_4 = \{12, 123, 124, 1234\}$ and $A_5 = \{\varnothing, 12, 123, 124, 1234\}$ all induce the same partition $P = \mathrm{par}(A_i) = \{12, 3, 4\} \in \mathrm{Part}(\Sigma)$. For example, $\alpha_{A_5}(\{1\}) = \alpha_{A_5}(\{2\}) = \{1, 2\}$, $\alpha_{A_5}(\{3\}) = \{1, 2, 3\}$ and $\alpha_{A_5}(\{4\}) = \{1, 2, 3, 4\}$ so that $\mathrm{par}(A_5) = P$. Observe that A_3 is the only partitioning abstract domain because $\mathrm{pad}(P) = A_3$.

Abstract domains of $\wp(\Sigma)$ carry additional information other than the underlying state partition and this additional information distinguishes them. As shown in [28], it turns out that this can be precisely stated by abstract interpretation since the above mappings par and pad allow us to view the whole lattice of partitions of Σ as a ("higher-order") abstraction of the lattice of abstract domains of $\wp(\Sigma)$:

$$(\mathrm{par}, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{Part}(\Sigma)_{\geq}, \mathrm{pad}) \text{ is a GI.}$$

As a consequence, the mappings par and pad give rise to an order isomorphism between state partitions and partitioning abstract domains: $\mathrm{Part}(\Sigma)_{\leq} \cong \mathrm{Abs}^{\mathrm{par}}(\wp(\Sigma))_{\sqsubseteq}$.

3.2. Abstract semantics and generalized strong preservation

Concrete semantics. We consider temporal specification languages $\mathcal{L}$ whose state formulae φ are inductively defined by:

$$\mathcal{L} \ni \varphi ::= p \mid f(\varphi_1, \ldots, \varphi_n)$$

where p ranges over a (typically finite) set of atomic propositions AP, while f ranges over a finite set Op of operators. AP and Op are also denoted, respectively, by $AP_{\mathcal{L}}$ and $Op_{\mathcal{L}}$. Each operator $f \in Op$ has an arity[3] $\sharp(f) > 0$.

Formulae in $\mathcal{L}$ are interpreted on a *semantic structure* $\mathcal{S} = (\Sigma, I)$ where Σ is any (possibly infinite) set of states and I is an interpretation function $I : AP \cup Op \to \mathrm{Fun}(\wp(\Sigma))$ that maps $p \in AP$ to some set $I(p) \in \wp(\Sigma)$ and $f \in Op$ to some function $I(f) : \wp(\Sigma)^{\sharp(f)} \to \wp(\Sigma)$. The interpretations $I(p)$ and $I(f)$ are also denoted by, respectively, $\boldsymbol{p}$ and $\boldsymbol{f}$. Moreover, $\boldsymbol{AP} \stackrel{\mathrm{def}}{=} \{\boldsymbol{p} \in \wp(\Sigma) \mid p \in AP\}$ and $\boldsymbol{Op} \stackrel{\mathrm{def}}{=} \{\boldsymbol{f} : \wp(\Sigma)^{\sharp(f)} \to \wp(\Sigma) \mid f \in Op\}$. The *concrete state semantic function* $[\![\cdot]\!]_{\mathcal{S}} : \mathcal{L} \to \wp(\Sigma)$ evaluates a formula $\varphi \in \mathcal{L}$ to the set of states making φ true with respect to the semantic structure $\mathcal{S}$:

$$[\![p]\!]_{\mathcal{S}} = \boldsymbol{p} \quad \text{and} \quad [\![f(\varphi_1, \ldots, \varphi_n)]\!]_{\mathcal{S}} = \boldsymbol{f}([\![\varphi_1]\!]_{\mathcal{S}}, \ldots, [\![\varphi_n]\!]_{\mathcal{S}}).$$

Semantic structures generalize the role of Kripke structures. In fact, in standard model checking a semantic structure is usually defined through a Kripke structure $\mathcal{K}$ so that the interpretation of logical/temporal operators is defined in terms of paths in $\mathcal{K}$ and standard logical operators. In the following, we freely use standard logical and temporal operators together with their usual interpretations: for example, $I(\wedge) = \cap$, $I(\vee) = \cup$, $I(\neg) = \complement$, and if $\to$ denotes a transition relation in $\mathcal{K}$ then $I(\mathrm{EX}) = \mathrm{pre}_{\to}$, $I(\mathrm{AX}) = \widetilde{\mathrm{pre}}_{\to}$, etc.

If g is any operator with arity $\sharp(g) = n > 0$, whose interpretation is given by $\boldsymbol{g} : \wp(\Sigma)^n \to \wp(\Sigma)$, and $\mathcal{S} = (\Sigma, I)$ is a semantic structure then we say that a language $\mathcal{L}$ is *closed under* g for $\mathcal{S}$ when for any $\varphi_1, \ldots, \varphi_n \in \mathcal{L}$ there exists some $\psi \in \mathcal{L}$ such that $\boldsymbol{g}([\![\varphi_1]\!]_{\mathcal{S}}, \ldots, [\![\varphi_n]\!]_{\mathcal{S}}) = [\![\psi]\!]_{\mathcal{S}}$. In particular, a language $\mathcal{L}$ is closed under (finite) infinite logical conjunction for $\mathcal{S}$ iff for any (finite) $\Phi \subseteq \mathcal{L}$, there exists some $\psi \in \mathcal{L}$ such that $\bigcap_{\varphi \in \Phi} [\![\varphi]\!]_{\mathcal{S}} = [\![\psi]\!]_{\mathcal{S}}$. In particular, let us note that if $\mathcal{L}$ is closed under infinite logical conjunction then it must exist some $\psi \in \mathcal{L}$ such that $\cap\varnothing = \Sigma = [\![\psi]\!]_{\mathcal{S}}$, namely $\mathcal{L}$ is able to express the tautology *true*. Let us also remark that if the state space Σ is finite and $\mathcal{L}$ is closed under logical conjunction then we also mean that there exists some $\psi \in \mathcal{L}$ such that $\cap\varnothing = \Sigma = [\![\psi]\!]_{\mathcal{S}}$. Finally, note that if $\mathcal{L}$ is closed under negation and (infinite) logical conjunction then $\mathcal{L}$ is closed under (infinite) logical disjunction as well.

Abstract semantics. Abstract interpretation allows to define abstract semantics. Let $\mathcal{L}$ be a language and $\mathcal{S} = (\Sigma, I)$ be a semantic structure for $\mathcal{L}$. An *abstract semantic structure* $\mathcal{S}^\sharp = (A, I^\sharp)$ is given by an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma)_{\subseteq})$ and by an abstract interpretation function $I^\sharp : AP \cup Op \to \mathrm{Fun}(A)$. An abstract semantic structure $\mathcal{S}^\sharp$ therefore induces an *abstract semantic function* $[\![\cdot]\!]_{\mathcal{S}^\sharp} : \mathcal{L} \to A$ that evaluates formulae in $\mathcal{L}$ to abstract values in A. In particular, the abstract domain A systematically induces an abstract semantic structure

[3] It would be possible to consider generic operators whose arity is any possibly infinite ordinal, thus allowing, for example, infinite conjunctions or disjunctions.

$\mathcal{S}^A = (A, I^A)$ where I^A is the best correct approximation of I on A, i.e., I^A interprets atoms p and operators f as best correct approximations on A of, respectively, $\boldsymbol{p}$ and $\boldsymbol{f}$: for any $p \in AP$ and $f \in Op$,

$$I^A(p) \stackrel{\text{def}}{=} \alpha(\boldsymbol{p}) \quad \text{and} \quad I^A(f) \stackrel{\text{def}}{=} \boldsymbol{f}^A = \alpha \circ \boldsymbol{f} \circ \langle \gamma, \ldots, \gamma \rangle.$$

Thus, the abstract domain A always induces an abstract semantic function $[\![\cdot]\!]_{\mathcal{S}^A} : \mathcal{L} \to A$, also denoted by $[\![\cdot]\!]^A_{\mathcal{S}}$, which is therefore defined by:

$$[\![p]\!]^A_{\mathcal{S}} = \alpha(\boldsymbol{p}) \quad \text{and} \quad [\![f(\varphi_1, \ldots, \varphi_n)]\!]^A_{\mathcal{S}} = \boldsymbol{f}^A([\![\varphi_1]\!]^A_{\mathcal{S}}, \ldots, [\![\varphi_n]\!]^A_{\mathcal{S}}).$$

Standard strong preservation. A state semantics $[\![\cdot]\!]_{\mathcal{S}}$, for a semantic/Kripke structure $\mathcal{S}$, induces a state logical equivalence $\equiv^{\mathcal{S}}_{\mathcal{L}} \subseteq \Sigma \times \Sigma$ as usual:

$$s \equiv^{\mathcal{S}}_{\mathcal{L}} s' \quad \text{iff} \quad \forall \varphi \in \mathcal{L}.\, s \in [\![\varphi]\!]_{\mathcal{S}} \Leftrightarrow s' \in [\![\varphi]\!]_{\mathcal{S}}.$$

Let $P_{\mathcal{L}} \in \text{Part}(\Sigma)$ be the partition induced by $\equiv^{\mathcal{S}}_{\mathcal{L}}$ (the index $\mathcal{S}$ denoting the semantic/Kripke structure is omitted). For a number of well-known temporal languages like CTL*, ACTL*, CTL*-X, it turns out that if a partition is more refined than $P_{\mathcal{L}}$ then it induces a standard *strongly preserving* (s.p.) abstract model. This means that if $\mathcal{L}$ is interpreted on a Kripke structure $\mathcal{K} = (\Sigma, \to, \ell)$ and $P \preceq P_{\mathcal{L}}$ then one can define an abstract Kripke structure $\mathcal{A} = (P, \to^\sharp, \ell^\sharp)$, having the partition P as abstract state space, that strongly preserves $\mathcal{L}$: for any $\varphi \in \mathcal{L}$, $s \in \Sigma$ and $B \in P$ such that $s \in B$, we have that $B \models^{\mathcal{A}} \varphi$ (that is, $B \in [\![\varphi]\!]_{\mathcal{A}}$) if and only if $s \models^{\mathcal{K}} \varphi$ (that is, $s \in [\![\varphi]\!]_{\mathcal{K}}$). Let us recall a couple of well-known examples (see e.g., [4,7]):

(i) Let $P_{\text{ACTL}^*} \in \text{Part}(\Sigma)$ be the partition induced by ACTL* on some Kripke structure $\mathcal{K} = (\Sigma, \to, \ell)$. If $P \preceq P_{\text{ACTL}^*}$ then the abstract Kripke structure $\mathcal{A} = (P, \to^{\forall\exists}, \ell_P)$ strongly preserves ACTL*, where $\ell_P(B) = \cup\{\ell(s) \mid s \in B\}$ and $\to^{\forall\exists} \subseteq P \times P$ is defined as: $B_1 \to^{\forall\exists} B_2 \Leftrightarrow \forall s_1 \in B_1.\, \exists s_2 \in B_2.\, s_1 \to s_2$.

(ii) Let $P_{\text{CTL}^*} \in \text{Part}(\Sigma)$ be the partition induced by CTL* on $\mathcal{K}$. If $P \preceq P_{\text{CTL}^*}$ then the abstract Kripke structure $\mathcal{A} = (P, \to^{\exists\exists}, \ell_P)$ strongly preserves CTL*, where $B_1 \to^{\exists\exists} B_2 \Leftrightarrow \exists s_1 \in B_1, s_2 \in B_2.\, s_1 \to s_2$.

Following Dams [7, Section 6.1] and Henzinger et al. [21, Section 2.2], the notion of strong preservation can be given with respect to a mere state partition rather than with respect to an abstract Kripke structure. A partition $P \in \text{Part}(\Sigma)$ is strongly preserving[4] for $\mathcal{L}$ (when interpreted on a semantic/Kripke structure $\mathcal{S}$) if $P \preceq P_{\mathcal{L}}$. In this sense, $P_{\mathcal{L}}$ is the coarsest partition that is strongly preserving for $\mathcal{L}$. For a number of well known temporal languages, like ACTL*, CTL* (see, respectively, the above points (i) and (ii)), CTL*-X and the fragments of the μ-calculus described by Henzinger et al. [21], it turns out that if P is strongly preserving for $\mathcal{L}$ then the abstract Kripke structure $(P, \to^{\exists\exists}, \ell_P)$ is strongly preserving for $\mathcal{L}$. In particular, $(P_{\mathcal{L}}, \to^{\exists\exists}, \ell_{P_{\mathcal{L}}})$ is strongly preserving for $\mathcal{L}$ and, additionally, $P_{\mathcal{L}}$ is the smallest possible abstract state space, namely if $\mathcal{A} = (A, \to^\sharp, \ell^\sharp)$ is an abstract Kripke structure that strongly preserves $\mathcal{L}$ then $|P_{\mathcal{L}}| \leqslant |A|$.

Generalized strong preservation. Intuitively, the partition $P_{\mathcal{L}}$ is an abstraction of the state semantics $[\![\cdot]\!]_{\mathcal{S}}$. Let us make this intuition precise. Following [28], an abstract domain $A \in \text{Abs}(\wp(\Sigma))$ is defined to be strongly preserving for $\mathcal{L}$ (with respect to $\mathcal{S}$) when for any $S \in \wp(\Sigma)$ and $\varphi \in \mathcal{L}$: $\alpha(S) \leqslant [\![\varphi]\!]^A_{\mathcal{S}} \Leftrightarrow S \subseteq [\![\varphi]\!]_{\mathcal{S}}$. This generalizes strong preservation from partitions to abstract domains because, by exploiting the isomorphism in Section 3.1 between partitions and partitioning abstract domains, it turns out that P is a s.p. partition for $\mathcal{L}$ with respect to $\mathcal{S}$ iff $\text{pad}(P)$ is a s.p. abstract domain for $\mathcal{L}$ with respect to $\mathcal{S}$.

[4] Dams [7] uses the term "fine" instead of "strongly preserving".

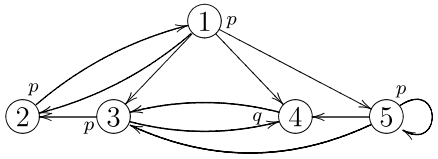

Fig. 2. A Kripke structure.

Forward complete shells and strong preservation. Partition refinement algorithms for computing behavioural equivalences like bisimulation [26], simulation equivalence [3,11,20,29,30] and (divergence blind) stuttering equivalence [17] are used in abstract model checking to compute the coarsest strongly preserving partition for temporal languages like CTL* or the μ-calculus for the case of bisimulation equivalence, ACTL* for simulation equivalence and CTL*-X for stuttering equivalence. Let us recall from [28] how the input/output behaviour of these partition refinement algorithms can be generalized through abstract interpretation. Given a language $\mathcal{L}$ and a concrete state space Σ, partition refinement algorithms work by iteratively refining an initial partition P within the lattice of partitions $\mathrm{Part}(\Sigma)$ until the fixpoint $P_{\mathcal{L}}$ is reached. The input partition P determines a set AP_P of atoms and a corresponding interpretation I_P as follows: $AP_P \stackrel{\text{def}}{=} \{p_B \mid B \in P\}$ and $I_P(p_B) \stackrel{\text{def}}{=} B$. More in general, any $\mathcal{X} \subseteq \wp(\Sigma)$ determines a set $\{p_X\}_{X\in\mathcal{X}}$ of atoms with interpretation $I_{\mathcal{X}}(p_X) = X$. In particular, this can be done for an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$ by considering its concretization $\gamma(A) \subseteq \Sigma$, namely A is viewed as a set of atoms $a \in A$ with interpretation $I_A(a) = \gamma(a)$. Thus, an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$ together with a set of functions $F \subseteq \mathrm{Fun}(\wp(\Sigma))$ determine a language $\mathcal{L}_{A,F}$, with atoms in A, operations in F and endowed with a semantic structure $\mathcal{S}_{A,F} = (\Sigma, I_A \cup I_F)$ such that for any $a \in A, I_A(a) = \gamma(a)$ and for any $f \in F, I_F(f) = f$. When $\mathcal{L}_{A,F}$ is closed under infinite logical conjunction (for finite state spaces this boils down to closure under finite conjunction) it turns out that the forward complete shell of A for F provides exactly the most abstract domain in $\mathrm{Abs}(\wp(\Sigma))$ that refines A and is strongly preserving for $\mathcal{L}_{A,F}$ (with respect to $\mathcal{S}_{A,F}$):

$$\mathscr{S}_F(A) = \sqcup\{X \in \mathrm{Abs}(\wp(\Sigma)) \mid X \sqsubseteq A, X \text{ is s.p. for } \mathcal{L}_{A,F}\}. \tag{3.1}$$

In other terms, forward complete shells coincide with strongly preserving shells.

On the other hand, let P_ℓ denote the state partition induced by the state labeling of a semantic/Kripke structure and let $\mathcal{L}$ be closed under logical conjunction and negation. Then, the coarsest s.p. partition $P_{\mathcal{L}}$ can be characterized as a forward complete shell as follows:

$$P_{\mathcal{L}} = \mathrm{par}(\mathscr{S}_{\boldsymbol{o}_{\boldsymbol{P}_{\mathcal{L}}}}(\mathrm{pad}(P_\ell))). \tag{3.2}$$

Example 3.2. Consider the following language $\mathcal{L}$:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EX}\varphi$$

and the Kripke structure $\mathcal{K}$ depicted in Fig. 2, where superscripts determine the labeling function ℓ and the interpretation of EX in $\mathcal{K}$ is the predecessor operator pre. The labeling function ℓ determines the partition $P_\ell = \{\boldsymbol{p} = 1235, \boldsymbol{q} = 4\} \in \mathrm{Part}(\Sigma)$, so that $\mathrm{pad}(P_\ell) = \{\varnothing, 1235, 4, 12345\} \in \mathrm{Abs}(\wp(\Sigma))$. Abstract domains are Moore-closed so that $\mathscr{S}_{\boldsymbol{o}_{\boldsymbol{P}_{\mathcal{L}}}} = \mathscr{S}_{\mathrm{pre}}$. Let us compute $\mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell))$.

$$X_0 = \mathrm{pad}(P_\ell) = \{\varnothing, 1235, 4, 12345\}$$

$$\begin{aligned} X_1 &= X_0 \sqcap \mathcal{M}(\mathrm{pre}(X_0)) = \mathcal{M}(X_0 \cup \mathrm{pre}(X_0)) \\ &= \mathcal{M}(\{\varnothing, 1235, 4, 12345\} \cup \{\mathrm{pre}(\{4\}) = 135\}) = \{\varnothing, 135, 1235, 4, 12345\} \end{aligned}$$

$$\begin{aligned} X_2 &= X_1 \sqcap \mathcal{M}(\mathrm{pre}(X_1)) = \mathcal{M}(X_1 \cup \mathrm{pre}(X_1)) \\ &= \mathcal{M}(\{\varnothing, 135, 1235, 4, 12345\} \cup \{\mathrm{pre}(\{135\}) = 1245\}) \\ &= \{\varnothing, 15, 125, 135, 1235, 4, 1245, 12345\} \end{aligned}$$

$$X_3 = X_2 \quad \text{(fixpoint)}.$$

By (3.1), X_2 is the most abstract domain that strongly preserves $\mathcal{L}$. Moreover, by (3.2), $P_{\mathcal{L}} = \mathrm{par}(X_2) = \{15, 2, 3, 4\}$ is the coarsest partition that strongly preserves $\mathcal{L}$. Observe that the abstract domain X_2 is not partitioning so that $\mathrm{pad}(P_{\mathcal{L}}) \sqsubset \mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell))$.

4. GPT: a generalized Paige–Tarjan refinement algorithm

In order to emphasize the ideas leading to our generalized Paige–Tarjan algorithm, let us first describe how some features of the Paige–Tarjan algorithm can be viewed and generalized from an abstract interpretation perspective.

4.1. A new perspective of PT

Consider a finite Kripke structure $(\Sigma, \rightarrow, \ell)$ over a set AP of atoms. In the following, $\mathrm{Part}(\Sigma)$ and $\mathrm{pre}_{\rightarrow}$ will be simply denoted by, respectively, Part and pre. As a direct consequence of (3.1), it turns out [28] that the output $\mathrm{PT}(P)$ of the Paige–Tarjan algorithm on an input partition $P \in \mathrm{Part}$ is the partitioning abstraction of the forward $\{\mathrm{pre}, \complement\}$-complete shell of $\mathrm{pad}(P)$, i.e.,

$$\mathrm{PT}(P) = \mathrm{par}(\mathscr{S}_{\{\mathrm{pre},\complement\}}(\mathrm{pad}(P))).$$

Hennessy–Milner logic HML is inductively generated by the logical/temporal operators of conjunction, negation and existential next, so that $\boldsymbol{Op}_{\mathrm{HML}} = \{\cap, \complement, \mathrm{pre}\}$. Moreover, as noted in Section 2.3, $\mathscr{S}_{\{\cap,\complement,\mathrm{pre}\}} = \mathscr{S}_{\{\complement,\mathrm{pre}\}}$. Hence, by (3.2), we observe that $\mathrm{PT}(P)$ computes the coarsest partition P_{HML} that is strongly preserving for HML.

On the other hand, equation (2.1) provides a constructive characterization of forward complete shells, meaning that it provides a naïve fixpoint algorithm for computing a complete shell $\mathscr{S}_F(A) = \mathrm{gfp}(F_A)$: begin with $X = \{\Sigma\} = \top_{\mathrm{Abs}(\wp(\Sigma))}$ and iteratively, at each step, compute $F_A(X)$ until a fixpoint is reached. This scheme could be in particular applied for computing $\mathscr{S}_{\{\mathrm{pre},\complement\}}(\mathrm{pad}(P))$. Note, however, that this naïve fixpoint algorithm is far from being efficient since at each step $F_A(X)$ always re-computes the images $f(\vec{x})$ that have already been computed at the previous step (cf. Example 2.1).

In our abstract interpretation view, PT is therefore an algorithm that computes

a particular abstraction of a particular forward complete shell.

Our goal is to analyze the basic steps of the PT algorithm in order to investigate whether it can be generalized from an abstract interpretation perspective to an algorithm that computes

a generic abstraction of a generic forward complete shell.

Let us first isolate in our framework the following key points concerning the PT algorithm.

Lemma 4.1. *Let* $P \in \mathrm{Part}$ *and* $S \subseteq \Sigma$.

(i) $\mathrm{PTsplit}(S, P) = \mathrm{par}(\mathcal{M}(\mathrm{pad}(P) \cup \{\mathrm{pre}(S)\})) = \mathrm{par}(\mathrm{pad}(P) \sqcap \mathcal{M}(\{\mathrm{pre}(S)\}))$.
(ii) $\mathrm{PTrefiners}(P) = \{S \in \mathrm{pad}(P) \mid \mathrm{par}(\mathcal{M}(\mathrm{pad}(P) \cup \{\mathrm{pre}(S)\})) \prec P\}$.
(iii) P *is* PT *stable iff* $\{S \in \mathrm{pad}(P) \mid \mathrm{par}(\mathcal{M}(\mathrm{pad}(P) \cup \{\mathrm{pre}(S)\})) \prec P\} = \varnothing$.

Proof. (i) By definition, $\mathrm{PTsplit}(S, P) = P \curlywedge \{\mathrm{pre}(S), \complement(\mathrm{pre}(S))\}$. Note that $\mathrm{par}(\mathcal{M}(\{\mathrm{pre}(S)\})) = \mathrm{par}(\{\mathrm{pre}(S), \Sigma\}) = \{\mathrm{pre}(S), \complement(\mathrm{pre}(S))\}$. Finally, observe that $\mathcal{M}(\mathrm{pad}(P) \cup \{\mathrm{pre}(S)\}) = \mathrm{pad}(P) \sqcap \mathcal{M}(\{\mathrm{pre}(S)\})$. Also, since the map $\mathrm{par} : \mathrm{Abs}_{\wp(\Sigma)})_{\sqsupseteq} \rightarrow \mathrm{Part}(\Sigma)_{\geq}$ is a left-adjoint and therefore it is additive, it turns out that

$$\begin{aligned}
\mathrm{par}(\mathcal{M}(\mathrm{pad}(P) \cup \{\mathrm{pre}(S)\})) &= && \text{[by the equation above]}\\
\mathrm{par}(\mathrm{pad}(P) \sqcap \mathcal{M}(\{\mathrm{pre}(S)\})) &= && \text{[by additivity of par]}\\
\mathrm{par}(\mathrm{pad}(P)) \curlywedge \mathrm{par}(\mathcal{M}(\{\mathrm{pre}(S)\})) &= && \text{[since par} \circ \mathrm{pad} = \mathrm{id}]\\
P \curlywedge \{\mathrm{pre}(S), \complement(\mathrm{pre}(S))\}. &&&
\end{aligned}$$

Points (ii) and (iii) follow immediately from (i). □

Given any set $S \subseteq \Sigma$, consider a domain refinement operation $\mathrm{refine}_{\mathrm{pre}}(S,\cdot) : \mathrm{Abs}(\wp(\Sigma)) \to \mathrm{Abs}(\wp(\Sigma))$ defined as

$$\mathrm{refine}_{\mathrm{pre}}(S,A) \stackrel{\mathrm{def}}{=} A \sqcap \mathcal{M}(\{\mathrm{pre}(S)\}) = \mathcal{M}(\gamma(A) \cup \{\mathrm{pre}(S)\}).$$

Observe that the best correct approximation of $\mathrm{refine}_{\mathrm{pre}}(S,\cdot)$ on the abstract domain Part is $\mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}(S,\cdot)$: $\mathrm{Part} \to \mathrm{Part}$ defined as

$$\mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}(S,P) \stackrel{\mathrm{def}}{=} \mathrm{par}(\mathrm{pad}(P) \sqcap \mathcal{M}(\{\mathrm{pre}(S)\})).$$

Thus, Lemma 4.1 (i) provides a characterization of the PT splitting step as best correct approximation of $\mathrm{refine}_{\mathrm{pre}}$ on Part. In turn, Lemma 4.1 (ii) and (iii) yield a characterization of PTrefiners and PT stability based on this best correct approximation $\mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}$. As a consequence, PT may be reformulated as follows.

while$(\{T \in \mathrm{pad}(P) \mid \mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}(T,P) \prec P\} \neq \varnothing)$ **do**
  choose $S \in \{T \in \mathrm{pad}(P) \mid \mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}(T,P) \prec P\}$;
  $P := \mathrm{refine}^{\mathrm{Part}}_{\mathrm{pre}}(S,P)$;

In the following, this view of PT is generalized to a generic abstract domain in $\mathrm{Abs}(\wp(\Sigma))$ in place of the partition P and to a generic set of operations on $\wp(\Sigma)$ in place of the predecessor pre.

4.2. Generalizing PT

Lemma 4.1 can be generalized as follows. Let $F \subseteq \mathrm{Fun}(\wp(\Sigma))$. We define a family of domain refinement operators $\mathrm{refine}_f : \wp(\Sigma)^{\sharp(f)} \to (\mathrm{Abs}(\wp(\Sigma)) \to \mathrm{Abs}(\wp(\Sigma)))$ indexed on functions $f \in F$ and tuples of sets $\vec{S} \in \wp(\Sigma)^{\sharp(f)}$:

(i) $\mathrm{Refine}_f(\vec{S},A) \stackrel{\mathrm{def}}{=} A \sqcap \mathcal{M}(\{f(\vec{S})\})$.

A tuple $\vec{S}$ is called a F-refiner for an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$ when there exists $f \in F$ such that $\vec{S} \in \gamma(A)^{\sharp(f)}$ and indeed $\vec{S}$ may contribute to refine A with respect to f, i.e., $\mathrm{refine}_f(\vec{S},A) \sqsubset A$. We thus define refiners of an abstract domain as follows:

(ii) $\mathrm{Refiners}_f(A) \stackrel{\mathrm{def}}{=} \{\vec{S} \in \gamma(A)^{\sharp(f)} \mid \mathrm{refine}_f(\vec{S},A) \sqsubset A\}$,
$\mathrm{Refiners}_F(A) \stackrel{\mathrm{def}}{=} \cup_{f \in F} \mathrm{Refiners}_f(A)$,

and in turn abstract domain stability as follows:

(iii) A is F-stable $\Leftrightarrow$ $\mathrm{Refiners}_F(A) = \varnothing$.

Concrete PT. The above observations lead us to design the following PT-like algorithm called CPT_F (Concrete PT), parameterized by F, which takes as input an abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$ and computes the forward F-complete shell of A.

ALGORITHM: CPT_F
Input: abstract domain $A \in \mathrm{Abs}(\wp(\Sigma))$
while$(\mathrm{Refiners}_F(A) \neq \varnothing)$ **do**
  choose for some $f \in F$, $\vec{S} \in \mathrm{Refiners}_f(A)$;
  $A := \mathrm{refine}_f(\vec{S},A)$;
Output: A

Lemma 4.2. *Let* $A \in \mathrm{Abs}(\wp(\Sigma))$.

(i) *A is forward F-complete iff* $\mathrm{Refiners}_F(A) = \varnothing$.
(ii) *Let* Σ *be finite. Then,* CPT_F *always terminates and* $\mathrm{CPT}_F(A) = \mathscr{S}_F(A)$.

Proof. (i) Given $f \in F$, notice that $A = \mathrm{refine}_f(\vec{S}, A)$ iff $f(\vec{S}) \in \gamma(A)$. Hence, $\mathrm{Refiners}_f(A) = \varnothing$ iff for any $\vec{S} \in \gamma(A)^{\sharp(f)}$, $f(\vec{S}) \in \gamma(A)$, namely, iff $f(\gamma(A)) \subseteq \gamma(A)$ iff A is forward f-complete. Thus, $\mathrm{Refiners}_F(A) = \varnothing$ iff A is forward F-complete.
(ii) We denote by $X_i \in \mathrm{uco}(\wp(\Sigma))$, $f_i \in F$ and $\vec{S}_i \in \mathrm{Refiners}_{f_i}(\mu_i)$ the sequences of, respectively, uco's, functions in F and refiners that are iteratively computed in some run of $\mathrm{CPT}_F(A)$, where $X_0 = A$. Observe that $\{X_i\}$ is a decreasing chain in $\mathrm{uco}(\wp(\Sigma))_{\sqsubseteq}$, hence, since Σ is assumed to be finite, it turns out that $\{X_i\}$ is finite. We denote by X_{fin} the last uco in the sequence $\{X_i\}$, i.e., $\mathrm{CPT}_F(A) = X_{fin}$. Since $\mathrm{Refiners}_F(X_{fin}) = \varnothing$, by point (i), X_{fin} is forward F-complete, and therefore, from $X_{fin} \sqsubseteq A$, we obtain that $X_{fin} \sqsubseteq \mathscr{S}_F(A)$.
Let us show, by induction on i, that $X_i \sqsupseteq \mathscr{S}_F(A)$.

($i = 0$): Clearly, $X_0 = A \sqsupseteq \mathscr{S}_F(A)$.
($i + 1$): By inductive hypothesis and monotonicity of refine_{f_i}, it turns out that $X_{i+1} = \mathrm{refine}_{f_i}(\vec{S}_i, X_i) \sqsupseteq \mathrm{refine}_{f_i}(\vec{S}_i, \mathscr{S}_F(A))$. Moreover, by point (i), since $\mathscr{S}_F(A)$ is forward f-complete, we have that $\mathrm{refine}_{f_i}(\vec{S}_i, \mathscr{S}_F(A)) = \mathscr{S}_F(A)$.

Thus, we obtain that $X_{fin} = \mathscr{S}_F(A)$. □

Example 4.3. Let us illustrate CPT on the abstract domain $A = \{\varnothing, 2, 1234\}$ of Example 2.1.

$$
\begin{array}{ll}
X_0 = A = \{\varnothing, 2, 1234\} & S_0 = \{2\} \in \mathrm{Refiners}_{\mathrm{post}_R}(X_0) \\
X_1 = \mathcal{M}(X_0 \cup \{\mathrm{post}_R(S_0)\}) & \\
\quad = \mathcal{M}(X_0 \cup \{3\}) = \{\varnothing, 2, 3, 1234\} & S_1 = \{3\} \in \mathrm{Refiners}_{\mathrm{post}_R}(X_1) \\
X_2 = \mathcal{M}(X_1 \cup \{\mathrm{post}_R(S_1)\}) & \\
\quad = \mathcal{M}(X_1 \cup \{4\}) = \{\varnothing, 2, 3, 4, 1234\} & S_2 = \{1234\} \in \mathrm{Refiners}_{\mathrm{post}_R}(X_2) \\
X_3 = \mathcal{M}(X_2 \cup \{\mathrm{post}_R(S_2)\}) & \\
\quad = \mathcal{M}(X_2 \cup \{234\}) = \{\varnothing, 2, 3, 4, 234, 1234\} & S_3 = \{234\} \in \mathrm{Refiners}_{\mathrm{post}_R}(X_3) \\
X_4 = \mathcal{M}(X_3 \cup \{\mathrm{post}_R(S_3)\}) & \\
\quad = \mathcal{M}(X_3 \cup \{34\}) = \{\varnothing, 2, 3, 4, 34, 234, 1234\} & \Rightarrow \mathrm{Refiners}_{\mathrm{post}_R}(X_4) = \varnothing
\end{array}
$$

Let us note that while in Example 2.1 each step consists in computing the images of post_R for the sets belonging to the whole domain at the previous step and this gives rise to re-computations, here instead an image $f(S_i)$ is never computed twice because at each step we nondeterministically choose a refiner S and apply post_R to S.

Abstract PT. Our goal is to design an abstract version of CPT_F that works on a generic abstraction $\mathcal{A}$ of the lattice of abstract domains $\mathrm{Abs}(\wp(\Sigma))$. As recalled in Section 3.1, partitions can be viewed as a "higher-order" abstraction of abstract domains through the Galois insertion $(\mathrm{par}, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{Part}(\Sigma)_{\geq}, \mathrm{pad})$. This is a dual GI since both order relations in $\mathrm{Abs}(\wp(\Sigma))$ and $\mathrm{Part}(\Sigma)$ are reversed. This depends on the fact that we want to obtain a complete approximation of a forward complete shell, which, by (2.1), is a greatest fixpoint so that we need to approximate a greatest fixpoint computation "from above" instead of "from below" as it happens for a least fixpoint computation. We thus consider a Galois insertion $(\alpha, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathcal{A}_{\geqslant}, \gamma)$ of an abstract domain $\mathcal{A}_{\geqslant}$ into the dual lattice of abstract domains $\mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}$. The order relation of the abstract domain $\mathcal{A}$ is denoted by $\geqslant$ because this makes concrete and abstract ordering notations uniform. It is worth remarking that since we require a Galois insertion of $\mathcal{A}$ into the complete lattice $\mathrm{Abs}(\wp(\Sigma))$, by standard results on Galois insertions [6], $\mathcal{A}$ must necessarily be a complete lattice. For any $f \in F$, the best correct approximation $\mathrm{refine}_f^{\mathcal{A}} : \wp(\Sigma)^{\sharp(f)} \to (\mathcal{A} \to \mathcal{A})$ of refine_f on $\mathcal{A}$ is therefore defined as usual by:

(i) $\mathrm{Refine}_f^{\mathcal{A}}(\vec{S}, a) \stackrel{\mathrm{def}}{=} \alpha(\mathrm{refine}_f(\vec{S}, \gamma(a)))$.

Accordingly, abstract refiners and stability are defined as follows:

(ii) $\mathrm{Refiners}_f^{\mathcal{A}}(a) \stackrel{\mathrm{def}}{=} \{\vec{S} \in \gamma(a)^{\sharp(f)} \mid \mathrm{refine}_f^{\mathcal{A}}(\vec{S}, a) < a\}$,
$\mathrm{Refiners}_F^{\mathcal{A}}(a) \stackrel{\mathrm{def}}{=} \cup_{f \in F} \mathrm{Refiners}_f^{\mathcal{A}}(a)$;
(iii) an abstract object $a \in \mathcal{A}$ is F-stable $\Leftrightarrow$ $\mathrm{Refiners}_F^{\mathcal{A}}(a) = \varnothing$.

We may now define the following abstract version of the above algorithm CPT_F, called $\mathrm{GPT}_F^{\mathcal{A}}$ (Generalized PT), that is parameterized on the abstraction $\mathcal{A}$.

ALGORITHM: $\mathrm{GPT}_F^{\mathcal{A}}$

Input: abstract object $a \in \mathcal{A}$
while $(\mathrm{Refiners}_F^{\mathcal{A}}(a) \neq \varnothing)$ **do**
  choose for some $f \in F$, $\vec{S} \in \mathrm{Refiners}_f^{\mathcal{A}}(a)$;
  $a := \mathrm{refine}_f^{\mathcal{A}}(\vec{S}, a)$;
Output: a

$\mathrm{GPT}_F^{\mathcal{A}}(a)$ computes a sequence of abstract objects $\{a_i\}_{i \in \mathbb{N}}$ which is a decreasing chain in $\mathcal{A}_{\leqslant}$, namely $a_{i+1} < a_i$. Thus, in order to ensure termination of $\mathrm{GPT}_F^{\mathcal{A}}$ it is enough to consider an abstraction $\mathcal{A}$ such that $\langle \mathcal{A}, \leqslant \rangle$ satisfies the descending chain condition (DCC), i.e., every descending chain is eventually stationary. Furthermore, let us remark that correctness for $\mathrm{GPT}_F^{\mathcal{A}}$ means that for any input object $a \in \mathcal{A}$, $\mathrm{GPT}_F^{\mathcal{A}}(a)$ computes exactly the abstraction in $\mathcal{A}$ of the forward F-complete shell of the abstract domain $\gamma(a)$, that is

$$\mathrm{GPT}_F^{\mathcal{A}}(a) = \alpha(\mathscr{S}_F(\gamma(a))).$$

Note that, by (2.1), $\alpha(\mathscr{S}_F(\gamma(a))) = \alpha(\mathrm{gfp}(F_{\gamma(a)}))$. It should be clear that correctness for GPT is somehow related to backward completeness in abstract interpretation. In fact, if the abstraction $\mathcal{A}$ is backward complete for $F_{\gamma(a)} = \lambda X.\gamma(a) \sqcap F^{\mathcal{M}}(X)$ then it is also fixpoint complete (cf. Section 2.2.2), so that $\alpha(\mathrm{gfp}(F_{\gamma(a)})) = \mathrm{gfp}(F_{\gamma(a)}^{\mathcal{A}})$, where $F_{\gamma(a)}^{\mathcal{A}}$ is the best correct approximation of the operator $F_{\gamma(a)}$ on the abstraction $\mathcal{A}$. The intuition is that $\mathrm{GPT}_F^{\mathcal{A}}(a)$ is an algorithm for computing the greatest fixpoint $\mathrm{gfp}(F_{\gamma(a)}^{\mathcal{A}})$. Indeed, the following result shows that $\mathrm{GPT}_F^{\mathcal{A}}$ is correct when $\mathcal{A}$ is backward complete for $F^{\mathcal{M}}$, because this implies that $\mathcal{A}$ is backward complete for F_A, for any abstract domain A. Moreover, we also isolate the following condition ensuring the correctness of $\mathrm{GPT}_F^{\mathcal{A}}$: the forward F-complete shell operator $\mathscr{S}_F$ maps domains in $\mathcal{A}$ into domains in $\mathcal{A}$, namely the higher-order abstraction $\mathcal{A}$ is forward complete for the forward F-complete shell $\mathscr{S}_F$.

Theorem 4.4. *Let $\mathcal{A}_{\leqslant}$ be DCC and assume that one of the following conditions holds*:

(i) *$\mathcal{A}$ is backward complete for $F^{\mathcal{M}}$,*
(ii) *$\mathcal{A}$ is forward complete for $\mathscr{S}_F$.*

Then, $\mathrm{GPT}_F^{\mathcal{A}}$ *always terminates and for any* $a \in \mathcal{A}$, $\mathrm{GPT}_F^{\mathcal{A}}(a) = \alpha(\mathscr{S}_F(\gamma(a)))$.

Proof. Let us first show the following two facts. For any $a \in \mathcal{A}$:

(A) $\mathrm{Refiners}_F(\gamma(a)) = \mathrm{Refiners}_F^{\mathcal{A}}(a)$;
(B) $\gamma(a)$ is forward F-complete iff $\mathrm{Refiners}_F^{\mathcal{A}}(a) = \varnothing$.

(A) Let $f \in F$. Note that $\mathrm{refine}_f(\vec{S}, \gamma(a)) = \gamma(a) \sqcap \mathcal{M}(\{f(\vec{S})\})$ and therefore $\mathrm{refine}_f^{\mathcal{A}}(\vec{S}, a) = \alpha(\gamma(a) \sqcap \mathcal{M}(\{f(\vec{S})\})) = \alpha(\gamma(a)) \wedge_A \alpha(\mathcal{M}(\{f(\vec{S})\})) = a \wedge_A \alpha(\mathcal{M}(\{f(\vec{S})\}))$. Consequently, $\vec{S} \in \mathrm{Refiners}_f(\gamma(a))$ iff $\vec{S} \in \gamma(a)^{\sharp(f)}$ and $\mathcal{M}(\{f(\vec{S})\}) \not\sqsupseteq \gamma(a)$. Likewise, we have that $\vec{S} \in \mathrm{Refiners}_f^{\mathcal{A}}(a)$ iff $\vec{S} \in \gamma(a)^{\sharp(f)}$ and $\alpha(\mathcal{M}(\{f(\vec{S})\})) \not\geqslant a$. These are equivalent properties, because, by Galois insertion, we have that $\alpha(\mathcal{M}(\{f(\vec{S})\})) \geqslant a$ iff $\mathcal{M}(\{f(\vec{S})\}) \sqsupseteq \gamma(a)$.

(B) $\gamma(a)$ is forward F-complete iff $\mathrm{Refiners}_F(\gamma(a)) = \varnothing$ iff $\mathrm{Refiners}_F^{\mathcal{A}}(a) = \varnothing$, by point (A).

Let us now prove the main result. We denote by $a_i \in \mathcal{A}$, $f_i \in F$ and $\vec{S}_i \in \mathrm{Refiners}_{f_i}^{\mathcal{A}}(a_i)$ the sequences of, respectively, abstract ojects, functions in F and refiners that are iteratively computed in some run of $\mathrm{GPT}_F^{\mathcal{A}}(a)$, where $a_0 = a$. Since $\{a_i\}$ is a decreasing chain in the abstract domain $\mathcal{A}_{\leqslant}$ which is assumed to be DCC, it turns out that these sequences are finite. We denote by a_{fin} the last element in the sequence of a_i's, i.e., $\mathrm{GPT}_F^{\mathcal{A}}(a) = a_{fin}$. Moreover, we also consider the following sequence of abstract domains: $X_i \stackrel{\mathrm{def}}{=} \gamma(a_i) \sqcap F^{\mathcal{M}}(\gamma(a_i)) = \mathcal{M}(\gamma(a_i) \cup F(\gamma(a_i)))$. Let us notice that, since $a_{i+1} \leqslant a_i$, by monotonicity, we have that $X_{i+1} \sqsubseteq X_i$. Moreover, since $\mathrm{Refiners}_F^{\mathcal{A}}(a_{fin}) = \varnothing$, by point (B), $\gamma(a_{fin})$ is forward F-complete, hence $\gamma(a_{fin}) \sqsubseteq F^{\mathcal{M}}(\gamma(a_{fin}))$, so that $X_{fin} = \gamma(a_{fin})$. We show that $\alpha(X_{fin}) = \alpha(\mathscr{S}_F(\gamma(a)))$, so that $a_{fin} = \alpha(\gamma(a_{fin})) = \alpha(X_{fin}) = \alpha(\mathscr{S}_F(\gamma(a)))$ follows. By point (A), $\mathrm{Refiners}_F(\gamma(a_{fin})) = \mathrm{Refiners}_F^{\mathcal{A}}(a_{fin}) = \varnothing$, thus, by Lemma 4.2 (i), $\gamma(a_{fin})$ is forward F-complete. Moreover, $\gamma(a_{fin}) \sqsubseteq \gamma(a_0) = \gamma(a)$ and consequently $\gamma(a_{fin}) \sqsubseteq \mathscr{S}_F(\gamma(a))$. Hence, $\alpha(X_{fin}) = \alpha(\gamma(a_{fin})) \leqslant \alpha(\mathscr{S}_F(\gamma(a)))$. Let us now show, by induction on i, that $\alpha(X_i) \geqslant \alpha(\mathscr{S}_F(\gamma(a)))$.

($i = 0$): $X_0 = \gamma(a_0) \sqcap F^{\mathcal{M}}(\gamma(a_0)) = \gamma(a) \sqcap F^{\mathcal{M}}(\gamma(a))$, hence, since $\mathscr{S}_F(\gamma(a)) \sqsubseteq \gamma(a), F^{\mathcal{M}}(\gamma(a))$, we have that $\mathscr{S}_F(\gamma(a)) \sqsubseteq X_0$, and therefore $\alpha(\mathscr{S}_F(\gamma(a))) \leqslant \alpha(X_0)$.

($i + 1$): Since $a_{i+1} = \alpha(\mathcal{M}(\gamma(a_i) \cup \{f_i(\vec{S}_i)\}))$, where $\vec{S}_i \in \gamma(a_i)$, we have that $f_i(\vec{S}_i) \in F^{\mathcal{M}}(\gamma(a_i))$. Hence, $\mathcal{M}(\gamma(a_i) \cup \{f_i(\vec{S}_i)\}) \subseteq \mathcal{M}(\gamma(a_i) \cup F^{\mathcal{M}}(\gamma(a_i))) = \gamma(a_i) \sqcap F^{\mathcal{M}}(\gamma(a_i)) = X_i$, namely $X_i \sqsubseteq \mathcal{M}(\gamma(a_i) \cup \{f_i(\vec{S}_i)\})$, so that $\alpha(X_i) \leqslant a_{i+1}$ and $\gamma(\alpha(X_i)) \sqsubseteq \gamma(a_{i+1})$. Moreover:

$$
\begin{aligned}
\alpha(X_{i+1}) &= \\
\alpha(\gamma(a_{i+1}) \sqcap F^{\mathcal{M}}(\gamma(a_{i+1}))) &= && \text{[since } \alpha \text{ is co-additive]} \\
\alpha(\gamma(a_{i+1})) \sqcap \alpha(F^{\mathcal{M}}(\gamma(a_{i+1}))) &\geqslant && \text{[since } \gamma(a_{i+1}) \sqsupseteq \gamma(\alpha(X_i))\text{]} \\
\alpha(\gamma(\alpha(X_i))) \sqcap \alpha(F^{\mathcal{M}}(\gamma(\alpha(X_i)))) &\geqslant && \text{[by induction]} \\
\alpha(\gamma(\alpha(\mathscr{S}_F(\gamma(a))))) \sqcap \alpha(F^{\mathcal{M}}(\gamma(\alpha(\mathscr{S}_F(\gamma(a))))))) &= && \text{[since } \alpha \circ \gamma \circ \alpha = \alpha\text{]} \\
\alpha(\mathscr{S}_F(\gamma(a))) \sqcap \alpha(\gamma(\alpha(F^{\mathcal{M}}(\gamma(\alpha(\mathscr{S}_F(\gamma(a)))))))).
\end{aligned}
$$

Now, both conditions (i) and (ii) imply that

$$\alpha(\gamma(\alpha(F^{\mathcal{M}}(\gamma(\alpha(\mathscr{S}_F(\gamma(a))))))))) = \alpha(\gamma(\alpha(F^{\mathcal{M}}(\mathscr{S}_F(\gamma(a))))))).$$

Thus, we may proceed as follows:

$$
\begin{aligned}
\alpha(\mathscr{S}_F(\gamma(a))) \sqcap \alpha(\gamma(\alpha(F^{\mathcal{M}}(\rho_A(\mathscr{S}_F(\gamma(a))))))) &= && \text{[by either condition (i) or (ii)]} \\
\alpha(\mathscr{S}_F(\gamma(a))) \sqcap \alpha(\gamma(\alpha(F^{\mathcal{M}}(\mathscr{S}_F(\gamma(a))))))) &= && \text{[since } \alpha \circ \gamma \circ \alpha = \alpha\text{]} \\
\alpha(\mathscr{S}_F(\gamma(a))) \sqcap \alpha(F^{\mathcal{M}}(\mathscr{S}_F(\gamma(a))))) &= && \text{[as } \mathscr{S}_F(\gamma(a)) \text{ is forward } F\text{-complete]} \\
\alpha(\mathscr{S}_F(\gamma(a))) \sqcap \alpha(\mathscr{S}_F(\gamma(a))) &= \\
\alpha(\mathscr{S}_F(\gamma(a))).
\end{aligned}
$$

Thus, this closes the proof. □

Corollary 4.5. *Under the hypotheses of Theorem 4.4, for any* $a \in \mathcal{A}$, $\mathrm{GPT}_F^{\mathcal{A}}(a)$ *is the* F*-stable shell of* a.

Proof. By Theorem 4.4, $\mathrm{GPT}_F^{\mathcal{A}}(a) \leqslant a$ and is F-stable. Let us show that $\mathrm{GPT}_F^{\mathcal{A}}(a)$ is indeed the F-stable shell of a. Let $b \in A$ such that $b \leqslant a$ and $\mathrm{Refiners}_F^{\mathcal{A}}(b) = \varnothing$. Since $b \leqslant a$, we have that $\gamma(b) \sqsubseteq \gamma(a)$. Moreover, by point (A) in the proof of Theorem 4.4, $\mathrm{Refiners}_F(\gamma(b)) = \mathrm{Refiners}_F^{\mathcal{A}}(b) = \varnothing$, so that $\gamma(b)$ is forward F-complete by Lemma 4.2 (i). Hence, $\gamma(b) \sqsubseteq \mathscr{S}_F(\gamma(a))$ and thus, by Theorem 4.4, $b = \alpha(\gamma(b)) \leqslant \alpha(\mathscr{S}_F(\gamma(a))) = \mathrm{GPT}_F^{\mathcal{A}}(a)$. □

Example 4.6. Let us consider again Examples 2.1 and 4.3. Recall from Section 2.3 that the disjunctive shell $\mathscr{S}_{\mathrm{dis}} : \mathrm{Abs}(\wp(\Sigma)) \to \mathrm{dAbs}(\wp(\Sigma))$ maps any abstract domain A to its disjunctive completion $\mathscr{S}_{\mathrm{dis}}(A) = \{\cup S \mid S \subseteq \gamma(A)\}$. It turns out that the disjunctive shell $\mathscr{S}_{\mathrm{dis}}$ allows to view $\mathrm{dAbs}(\wp(\Sigma))_{\sqsupseteq}$ as an abstraction of $\mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}$, namely $(\mathscr{S}_{\mathrm{dis}}, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{dAbs}(\wp(\Sigma))_{\sqsupseteq}, id)$ is a GI. This is a consequence of the fact that disjunctive abstract domains are closed under lub's in $\mathrm{Abs}(\wp(\Sigma))$ and therefore $\mathrm{dAbs}(\wp(\Sigma))_{\sqsupseteq}$ is a Moore-family of $\mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}$. It turns out that condition (i) of Theorem 4.4 is satisfied for this GI. In fact, by exploiting the fact that $\mathrm{post}_R : \wp(\Sigma) \to \wp(\Sigma)$ is additive, it is not hard to verify that $\mathscr{S}_{\mathrm{dis}} \circ \mathrm{post}_R^{\mathcal{M}} \circ \mathscr{S}_{\mathrm{dis}} = \mathscr{S}_{\mathrm{dis}} \circ \mathrm{post}_R^{\mathcal{M}}$. Thus, let us apply $\mathrm{GPT}_{\mathrm{post}_R}^{\mathrm{dAbs}}$ to the disjunctive abstract domain $X_0 = \{\varnothing, 2, 1234\} = \mathscr{S}_{\mathrm{dis}}(\{2, 1234\}) \in \mathrm{dAbs}(\wp(\Sigma))$.

$$X_0 = \{\varnothing, 2, 1234\} \qquad S_0 = \{2\} \in \mathrm{Refiners}_{\mathrm{post}_R}^{\mathrm{dAbs}}(X_0)$$

$$\begin{aligned} X_1 &= \mathscr{S}_{\mathrm{dis}}(\mathcal{M}(X_0 \cup \{\mathrm{post}_R(S_0)\})) \\ &= \mathscr{S}_{\mathrm{dis}}(\{\varnothing, 2, 3, 1234\}) \\ &= \{\varnothing, 2, 3, 23, 1234\} \qquad S_1 = \{3\} \in \mathrm{Refiners}_{\mathrm{post}_R}^{\mathrm{dAbs}}(X_1) \end{aligned}$$

$$\begin{aligned} X_2 &= \mathscr{S}_{\mathrm{dis}}(\mathcal{M}(X_1 \cup \{\mathrm{post}_R(S_1)\})) \\ &= \mathscr{S}_{\mathrm{dis}}(\{\varnothing, 2, 3, 23, 4, 1234\}) \\ &= \{\varnothing, 2, 3, 4, 23, 24, 34, 234, 1234\} \quad \Rightarrow \mathrm{Refiners}_{\mathrm{post}_R}^{\mathrm{dAbs}}(X_2) = \varnothing \end{aligned}$$

From Example 4.3 we know that $\mathscr{S}_{\mathrm{post}_R}(X_0) = \{\varnothing, 2, 3, 4, 34, 234, 1234\}$. Thus, as expected from Theorem 4.4, $\mathrm{GPT}_{\mathrm{post}_R}^{\mathrm{dAbs}}(X_0)$ coincides with $\mathscr{S}_{\mathrm{dis}}(\mathscr{S}_{\mathrm{post}_R}(X_0)) = \{\varnothing, 2, 3, 4, 23, 24, 34, 234, 1234\}$. Note that the abstract fixpoint has been reached in two iterations, whereas in Example 4.3 the concrete computation by $\mathrm{CPT}_{\mathrm{post}_R}$ needed four iterations. □

4.3. An optimization of GPT

As pointed out by Paige and Tarjan [26], the PT algorithm works even if splitters are chosen among blocks instead of unions of blocks, i.e., if $\mathrm{PTrefiners}(P)$ is replaced with the subset of "block refiners" $\mathrm{PTblockrefiners}(P) \stackrel{\mathrm{def}}{=} \mathrm{PTrefiners}(P) \cap P$. This can be easily generalized as follows. Given $g \in F$, for any $a \in \mathcal{A}$, let $\mathrm{subRefiners}_g^{\mathcal{A}}(a) \subseteq \mathrm{Refiners}_g^{\mathcal{A}}(a)$ be any subset of refiners. We denote by $\mathrm{IGPT}_F^{\mathcal{A}}$ (which stands for Improved GPT) the version of $\mathrm{GPT}_F^{\mathcal{A}}$ where $\mathrm{Refiners}_g^{\mathcal{A}}$ is replaced with $\mathrm{subRefiners}_g^{\mathcal{A}}$. If stability for subrefiners is equivalent to stability for refiners then IGPT results to be correct.

Corollary 4.7. *Let* $g \in F$ *be such that, for any* $a \in \mathcal{A}$, $\mathrm{subRefiners}_g^{\mathcal{A}}(a) = \varnothing \Leftrightarrow \mathrm{Refiners}_g^{\mathcal{A}}(a) = \varnothing$. *Then, for any* $a \in \mathcal{A}$, $\mathrm{GPT}_F^{\mathcal{A}}(a) = \mathrm{IGPT}_F^{\mathcal{A}}(a)$.

Proof. Let $\mathrm{subRefiners}_F^{\mathcal{A}}(a) = \mathrm{subRefiners}_g^{\mathcal{A}}(a) \cup (\cup_{F \ni f \neq g} \mathrm{Refiners}_f^{\mathcal{A}}(a))$. By hypothesis, we have that $\mathrm{subRefiners}_F^{\mathcal{A}}(a) \neq \varnothing$ iff $\mathrm{Refiners}_F^{\mathcal{A}}(a) \neq \varnothing$. Let $\{a_i\}$ be the finite decreasing chain of abstract objects computed by $\mathrm{IGPT}_F^{\mathcal{A}}(a)$. Since $\mathrm{subRefiners}_F^{\mathcal{A}}(\mathrm{IGPT}_F^{\mathcal{A}}(a)) = \varnothing$ we have that $\mathrm{Refiners}_F^{\mathcal{A}}(\mathrm{IGPT}_F^{\mathcal{A}}(a)) = \varnothing$. Moreover, since, for any i, $\mathrm{subRefiners}_g^{\mathcal{A}}(a_i) \subseteq \mathrm{Refiners}_g^{\mathcal{A}}(a_i)$, there exists a run of $\mathrm{GPT}_F^{\mathcal{A}}(a)$ which exactly computes the sequence $\{a_i\}$, so that, by Theorem 4.4, $\mathrm{IGPT}_F^{\mathcal{A}}(a) = \mathrm{GPT}_F^{\mathcal{A}}(a)$. □

4.4. Instantiating GPT with partitions

Let us now show how the above GPT algorithm can be instantiated to the lattice of partitions. Assume that the state space Σ is finite. Recall from Section 3 that the lattice of partitions can be viewed as an approximation of the lattice of abstract domains through the GI $(\mathrm{par}, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{Part}(\Sigma)_{\geq}, \mathrm{pad})$. The following properties (1) and (2) are consequences of the fact that a partitioning abstract domain $\mathrm{pad}(P)$ is closed under complements, i.e., $X \in \mathrm{pad}(P)$ iff $\complement(X) \in \mathrm{pad}(P)$.

(1) $\mathrm{Refiners}^{\mathrm{Part}}_{\complement}(P) = \varnothing$.
(2) For any f and $\vec{S} \in \wp(\Sigma)^{\sharp(f)}$, $\mathrm{refine}^{\mathrm{Part}}_{f}(\vec{S}, P) = P \curlywedge \{f(\vec{S}), \complement(f(\vec{S}))\}$.

Thus, by Point (1), for any $F \subseteq \mathrm{Fun}(\wp(\Sigma))$, a partition $P \in \mathrm{Part}(\Sigma)$ is F-stable iff P is $(F \cup \{\complement\})$-stable, that is complements can be left out. Hence, if $F^{-\complement}$ denotes $F \setminus \{\complement\}$ then $\mathrm{GPT}^{\mathrm{Part}}_{F}$ may be simplified as follows.

ALGORITHM: $\mathrm{GPT}^{\mathrm{Part}}_{F}$

Input: partition $P \in \mathrm{Part}(\Sigma)$
while $(\mathrm{Refiners}^{\mathrm{Part}}_{F^{-\complement}}(P) \neq \varnothing)$ **do**
  choose for some $f \in F^{-\complement}$, $\vec{S} \in \mathrm{Refiners}^{\mathrm{Part}}_{f}(P)$;
  $P := P \curlywedge \{f(\vec{S}), \complement(f(\vec{S}))\}$;
Output: P

Note that the number of iterations of $\mathrm{GPT}^{\mathrm{Part}}_{F}$ is bounded by the height of the lattice $\mathrm{Part}(\Sigma)$, namely by the number of states $|\Sigma|$. Thus, if each refinement step involving some $f \in F$ takes $O(\mathrm{cost}(f))$-time then the time complexity of $\mathrm{GPT}^{\mathrm{Part}}_{F}$ is bounded by $O(|\Sigma| \max(\{\mathrm{cost}(f) \mid f \in F\}))$.

Let us now consider a language $\mathcal{L}$ and a semantic structure (Σ, I) for $\mathcal{L}$. If $\mathcal{L}$ is closed under logical conjunction and negation then, for any $A \in \mathrm{Abs}(\wp(\Sigma))$, $\mathscr{S}_{\mathit{Op}_{\mathcal{L}}}(A)$ is closed under complements and therefore it is a partitioning abstract domain. Thus, condition (ii) of Theorem 4.4 is satisfied since $\mathscr{S}_{\mathit{Op}_{\mathcal{L}}}$ maps partitioning abstract domains into partitioning abstract domains. The following characterization is thus obtained as a consequence of (3.2).

Corollary 4.8. *If $\mathcal{L}$ is closed under conjunction and negation then* $\mathrm{GPT}^{\mathrm{Part}}_{\mathit{Op}_{\mathcal{L}}}(P_{\ell}) = P_{\mathcal{L}}$.

This provides an algorithm parameterized on a language $\mathcal{L}$ that includes propositional logic for computing the coarsest strongly preserving partition $P_{\mathcal{L}}$.

PT as an instance of GPT. It is now immediate to obtain PT as an instance of GPT. We know that $\mathrm{GPT}^{\mathrm{Part}}_{\{\mathrm{pre},\complement\}} = \mathrm{GPT}^{\mathrm{Part}}_{\mathrm{pre}}$. Moreover, by Lemma 4.1 (i) and (ii):

$$P \curlywedge \{\mathrm{pre}(S), \complement(\mathrm{pre}(S))\} = \mathrm{PTsplit}(S, P) \text{ and } \mathrm{Refiners}^{\mathrm{Part}}_{\mathrm{pre}}(P) = \mathrm{PTrefiners}(P).$$

Hence, by Lemma 4.1 (iii), it turns out that $P \in \mathrm{Part}(\Sigma)$ is PT stable if and only if $\mathrm{Refiners}^{\mathrm{Part}}_{\mathrm{pre}}(P) = \varnothing$. Thus, the instance $\mathrm{GPT}^{\mathrm{Part}}_{\mathrm{pre}}$ provides *exactly* the PT algorithm. Also, correctness follows from Corollaries 4.5 and 4.8: $\mathrm{GPT}^{\mathrm{Part}}_{\mathrm{pre}}(P)$ is both the coarsest PT stable refinement of P and the coarsest strongly preserving partition P_{HML}.

5. Applications

5.1. Stuttering equivalence and Groote–Vaandrager algorithm

Lamport's criticism [24] of the next-time operator X in CTL/CTL* is well known. This motivated the study of temporal logics like CTL-X/CTL*-X obtained from CTL/CTL* by removing the next-time operator and led to

study a notion of *stuttering*-based equivalence in Kripke structures [2,8,17]. We are interested here in *divergence blind stuttering* (dbs for short) equivalence. Let $\mathcal{K} = (\Sigma, \rightarrow, \ell)$ be a Kripke structure over a set AP of atoms. A relation $R \subseteq \Sigma \times \Sigma$ is a divergence blind stuttering relation on $\mathcal{K}$ if for any $s, s' \in \Sigma$ such that sRs':

(1) $\ell(s) = \ell(s')$;
(2) If $s \rightarrow t$ then there exist $t_0, \ldots, t_k \in \Sigma$, with $k \geqslant 0$, such that: (i) $t_0 = s'$; (ii) for all $i \in [0, k-1]$, $t_i \rightarrow t_{i+1}$ and sRt_i; (iii) tRt_k;
(3) $s'Rs$, i.e., R is symmetric.

Observe that condition (2) allows the case $k = 0$ and this simply boils down to requiring that tRs'. It turns out that the empty relation is a dbs relation and that dbs relations are closed under union. Hence, the largest dbs relation exists and is an equivalence relation called dbs equivalence, whose corresponding partition is denoted by $P_{\mathrm{dbs}} \in \mathrm{Part}(\Sigma)$.

It turns out [8] that P_{dbs} (see also [28, Corollary 7.4]) can be characterized as the coarsest strongly preserving partition $P_{\mathcal{L}}$ for the following language $\mathcal{L}$:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EU}(\varphi_1, \varphi_2),$$

where the semantics $\mathbf{EU} : \wp(\Sigma)^2 \rightarrow \wp(\Sigma)$ of the existential until operator EU is as usual:

$$\mathbf{EU}(S_1, S_2) = S_2 \cup \{s \in S_1 \mid \exists s_0, \ldots, s_n \in \Sigma, \text{ with } n \geqslant 0, \text{ such that (i) } s_0 = s, \text{ (ii) } \forall i \in [0, n).\, s_i \in S_1,\ s_i \rightarrow s_{i+1},\ \text{(iii) } s_n \in S_2\}.$$

Therefore, as a straight instance of Corollary 4.8, it turns out that $\mathrm{GPT}^{\mathrm{Part}}_{\mathbf{EU}}(P_\ell) = P_{\mathcal{L}} = P_{\mathrm{dbs}}$.

Groote and Vaandrager [17] designed a partition refinement algorithm, here denoted by GV, for computing the partition P_{dbs}. This algorithm uses the following definitions of split and refiner:[5] For any $P \in \mathrm{Part}(\Sigma)$ and $B_1, B_2 \in P$,

$$\mathrm{GVsplit}(\langle B_1, B_2\rangle, P) \stackrel{\mathrm{def}}{=} P \between \{\mathbf{EU}(B_1, B_2), \complement(\mathbf{EU}(B_1, B_2))\},$$
$$\mathrm{GVrefiners}(P) \stackrel{\mathrm{def}}{=} \{\langle B_1, B_2\rangle \in P \times P \mid \mathrm{GVsplit}(\langle B_1, B_2\rangle, P) \prec P\}.$$

The algorithm GV is as follows. Groote and Vaandrager show how GV can be efficiently implemented in $O(|\rightarrow||\Sigma|)$-time.

ALGORITHM: GV
Input: partition $P \in \mathrm{Part}(\Sigma)$
while $(\mathrm{GVrefiners}(P) \neq \varnothing)$ **do**
  choose $\langle B_1, B_2\rangle \in \mathrm{GVrefiners}(P)$;
  $P := \mathrm{GVsplit}(\langle B_1, B_2\rangle, P)$;
Output: P

It turns out that GV exactly coincides with the optimized instance $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EU}}$ that considers block refiners. This is obtained as a straight consequence of the following facts.

Lemma 5.1

(1) $\mathrm{GVrefiners}(P) = \varnothing$ *iff* $\mathrm{Refiners}^{\mathrm{Part}}_{\mathbf{EU}}(P) = \varnothing$.
(2) $\mathrm{GVsplit}(\langle B_1, B_2\rangle, P) = \mathrm{refine}^{\mathrm{Part}}_{\mathbf{EU}}(\langle B_1, B_2\rangle, P)$.

[5] In [17], $pos(B_1, B_2)$ denotes $\mathbf{EU}(B_1, B_2) \cap B_1$.

Proof. (1) It is sufficient to show that if for any $B_1, B_2 \in P$, $\mathbf{EU}(B_1, B_2) \in \mathrm{pad}(P)$, then for any $S_1, S_2 \in \mathrm{pad}(P)$, $\mathbf{EU}(S_1, S_2) \in \mathrm{pad}(P)$. Thus, we have to prove that for any $\{B_i\}_{i\in I}, \{B_j\}_{j\in J} \subseteq P$, $\mathbf{EU}(\cup_i B_i, \cup_j B_j) = \cup_k B_k$, for some $\{B_k\}_{k\in K} \subseteq P$. $\mathbf{EU}$ is an additive operator in its second argument, thus we only need to show that, for any $B \in P$, $\mathbf{EU}(\cup_i B_i, B) = \cup_k B_k$, i.e., if $s \in \mathbf{EU}(\cup_i B_i, B)$ and $s \in B'$, for some $B' \in P$, then $B' \subseteq \mathbf{EU}(\cup_i B_i, B)$. If $s \in \mathbf{EU}(\cup_i B_i, B)$, for some $B \in P$, then there exist $n \geqslant 0$ and $s_0, ..., s_n \in \Sigma$ such that $s_0 = s$, for all $j \in [0, n-1]$, $s_j \in \cup_i B_i$ and $s_j \rightarrow s_{j+1}$, and $s_n \in B$. Let us prove by induction on n that if $s' \in B'$ then $s' \in \mathbf{EU}(\cup_i B_i, B)$.

- $n = 0$: In this case $s \in \cup_i B_i$ and $s \in B = B'$. Hence, for some k, $s \in B_k = B = B'$ and therefore $s \in \mathbf{EU}(B, B) = B$. Moreover, $\mathbf{EU}$ is monotone on its first argument and therefore $B' = B = \mathbf{EU}(B, B) \subseteq \mathbf{EU}(\cup_i B_i, B)$.
- $n + 1$: Suppose that there exist $s_0, ..., s_{n+1} \in \Sigma$ such that $s_0 = s$, $\forall j \in [0, n].s_j \in \cup_i B_i$ and $s_j \rightarrow s_{j+1}$, and $s_{n+1} \in B$. Let $s_n \in B_k$, for some $B_k \in \{B_i\}_{i\in I}$. Then, $s \in \mathbf{EU}(\cup_i B_i, B_k)$ and $s = s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$. Since this trace has length n, by inductive hypothesis, $s' \in \mathbf{EU}(\cup_i B_i, B_k)$. Hence, there exist $r_0, ..., r_m \in \Sigma$, with $m \geqslant 0$, such that $s' = r_0, \forall j \in [0, m-1].r_j \in \cup_i B_i$ and $r_j \rightarrow r_{j+1}$, and $r_m \in B_k$. Moreover, since $s_n \rightarrow s_{n+1}$, we have that $s_n \in \mathbf{EU}(B_k, B)$. By hypothesis, $\mathbf{EU}(B_k, B) \supseteq B_k$, and therefore $r_m \in \mathbf{EU}(B_k, B)$. Thus, there exist $q_0, ..., q_l \in \Sigma$, with $l \geqslant 0$, such that $r_m = q_0, \forall j \in [0, l-1].q_j \in B_k$ and $q_j \rightarrow q_{j+1}$, and $q_l \in B$. We have thus found the following trace: $s' = r_0 \rightarrow r_1 \rightarrow ... \rightarrow r_m = q_0 \rightarrow q_1 \rightarrow ... \rightarrow q_l$, where all the states in the sequence but the last one q_l belong to $\cup_i B_i$, while $q_l \in B$. This means that $s' \in \mathbf{EU}(\cup_i B_i, B)$.

(2) By Point (2) in Section 4.4, $\mathrm{refine}^{\mathrm{Part}}_{\mathbf{EU}}(\langle B_1, B_2\rangle, P) = P \between \{\mathbf{EU}(B_1, B_2), \complement(\mathbf{EU}(B_1, B_2))\} = \mathrm{GVsplit}(\langle B_1, B_2\rangle, P)$. □

Hence, by Corollary 4.7, we have that Lemma 5.1 (1) allows us to exploit the $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EU}}$ algorithm in order to choose refiners for $\mathbf{EU}$ among the pairs of blocks of the current partition, so that by Lemma 5.1 (2) we obtain that $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EU}}$ exactly coincides with the GV algorithm.

5.2. A new simulation equivalence algorithm

It is well known that simulation equivalence is an appropriate state equivalence to be used in abstract model checking because it strongly preserves ACTL* and provides a better state-space reduction than bisimulation equivalence. However, computing simulation equivalence is harder than bisimulation [23]. A number of algorithms for computing simulation equivalence exist: Henzinger, Henzinger and Kopke [20], Bloom and Paige [1], Bustan and Grumberg [3], Tan and Cleaveland [30], Gentilini, Piazza and Policriti [11] and Ranzato and Tapparo [29]. Let P_{sim} denote the partition corresponding to simulation equivalence so that $|P_{\mathrm{sim}}|$ is the number of simulation equivalence classes. The algorithms by Henzinger, Henzinger and Kopke [20] and Bloom and Paige [1] run in $O(|\rightarrow||\Sigma|)$-time and have the drawback of a quadratic space complexity that is limited from below by $O(|\Sigma|^2)$. A better space complexity is obtained by Gentilini, Piazza and Policriti's algorithm [11] that runs in $O(|P_{\mathrm{sim}}|^2 + |\Sigma|\log(|P_{\mathrm{sim}}|))$-space and $O(|\rightarrow||P_{\mathrm{sim}}|^2)$-time. On the other hand, the algorithm by Ranzato and Tapparo [29] runs in $O(|\rightarrow||P_{\mathrm{sim}}|)$-time and $O(|\Sigma||P_{\mathrm{sim}}|))$-space. As far as time-complexity is concerned, this latter is the best available algorithm and it still retains a space complexity which is comparable to that of Gentilini et al.'s algorithm. It is worth remarking that all these algorithms are quite sofisticated and may use complex data structures. We show how GPT can be instantiated in order to design a new simple and efficient simulation equivalence algorithm with competitive space and time complexities of, respectively, $O(|P_{\mathrm{sim}}|^2 + |\Sigma|\log(|P_{\mathrm{sim}}|))$ and $O(|P_{\mathrm{sim}}|^2 \cdot (|P_{\mathrm{sim}}|^2 + |\rightarrow|))$.

Consider a finite Kripke structure $\mathcal{K} = (\Sigma, \rightarrow, \ell)$. A relation $R \subseteq \Sigma \times \Sigma$ is a simulation on $\mathcal{K}$ if for any $s, s' \in \Sigma$ such that sRs':

(1) $\ell(s') \subseteq \ell(s)$;
(2) For any $t \in \Sigma$ such that $s \rightarrow t$, there exists $t' \in \Sigma$ such that $s' \rightarrow t'$ and tRt'.

Simulation equivalence $\sim_{\mathrm{sim}} \subseteq \Sigma \times \Sigma$ is defined as follows: $s \sim_{\mathrm{sim}} s'$ iff there exist two simulation relations R_1 and R_2 such that sR_1s' and $s'R_2s$. $P_{\mathrm{sim}} \in \mathrm{Part}(\Sigma)$ denotes the partition corresponding to $\sim_{\mathrm{sim}}$.

It is known (see e.g., [15, Section 8]) that simulation equivalence on $\mathcal{K}$ can be characterized as the state equivalence induced by the following language $\mathcal{L}$:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EX}\varphi$$

namely, $P_{\mathrm{sim}} = P_{\mathcal{L}}$, where the interpretation of EX in $\mathcal{K}$ is the standard predecessor operator. Let us consider the GI $(\mathscr{S}_{\mathrm{dis}}, \mathrm{Abs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{dAbs}(\wp(\Sigma))_{\sqsupseteq}, \mathrm{id})$ of disjunctive abstract domains into the lattice of abstract domains that we defined in Example 4.6. As observed in Example 4.6, it turns out that $\mathscr{S}_{\mathrm{dis}} \circ \mathrm{pre}^{\mathcal{M}} \circ \mathscr{S}_{\mathrm{dis}} = \mathscr{S}_{\mathrm{dis}} \circ \mathrm{pre}^{\mathcal{M}}$, namely the abstraction $\mathrm{dAbs}(\wp(\Sigma))$ is backward complete for $\mathrm{pre}^{\mathcal{M}}$. Thus, by applying Theorem 4.4 (i) we obtain

$$\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}(P_\ell) = \mathscr{S}_{\mathrm{dis}}(\mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell))).$$

In turn, by applying the partitioning abstraction par we obtain

$$\mathrm{par}(\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}(P_\ell)) = \mathrm{par}(\mathscr{S}_{\mathrm{dis}}(\mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell)))) = \mathrm{par}(\mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell)))$$

because $\mathrm{par} \circ \mathscr{S}_{\mathrm{dis}} = \mathrm{par}$. Also, by (3.2), we know that $\mathrm{par}(\mathscr{S}_{\mathrm{pre}}(\mathrm{pad}(P_\ell))) = P_{\mathcal{L}} = P_{\mathrm{sim}}$. We have therefore shown that

$$\mathrm{par}(\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}(P_\ell)) = P_{\mathrm{sim}}$$

namely the following instance $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ allows to compute simulation equivalence.

ALGORITHM: $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$

Input: disjunctive abstract domain $A := \mathscr{S}_{\mathrm{dis}}(\{[s]_\ell\}_{s\in\Sigma}) \in \mathrm{dAbs}(\wp(\Sigma))$
while $(\mathrm{Refiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A) \neq \varnothing)$ **do**
  choose $S \in \mathrm{Refiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A)$;
  $A := \mathrm{refine}^{\mathrm{dAbs}}_{\mathrm{pre}}(S, A)$;
Output: A

$\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ works by iteratively refining a disjunctive abstract domain $A \in \mathrm{dAbs}(\wp(\Sigma))$, which is first initialized to the disjunctive shell of the abstract domain that is determined by the labeling of atoms. Then, $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ iteratively finds a refiner S for A, namely a set $S \in \gamma(A)$ such that $\mathrm{pre}_{\rightarrow}(S)$ does not belong to $\gamma(A)$ and therefore may contribute to refine A, i.e., $\mathrm{refine}^{\mathrm{dAbs}}_{\mathrm{pre}}(S, A) = \mathbb{D}(\gamma(A) \cup \mathrm{pre}_{\rightarrow}(S)) \sqsubset A$. Simulation equivalence is then computed from the output disjunctive abstract domain A as $P_{\mathrm{sim}} = \mathrm{par}(A)$.

It turns out that refiners of a disjunctive abstract domain A can be chosen among the images of blocks in $\mathrm{par}(A)$, namely in

$$\mathrm{subRefiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A) \stackrel{\mathrm{def}}{=} \mathrm{Refiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A) \cap \{\gamma(\alpha(B)) \mid B \in \mathrm{par}(A)\}.$$

In fact, since both $\gamma \circ \alpha$ and $\mathrm{pre}_{\rightarrow}$ are additive functions, it turns out that $\forall S \in \gamma(A).\ \mathrm{pre}_{\rightarrow}(S) \in \gamma(A)$ if and only if $\forall B \in \mathrm{par}(A).\ \mathrm{pre}_{\rightarrow}(\gamma(\alpha(B))) \in \gamma(A)$, so that $\mathrm{subRefiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A) = \varnothing$ iff $\mathrm{Refiners}^{\mathrm{dAbs}}_{\mathrm{pre}}(A) = \varnothing$, and therefore Corollary 4.7 can be applied.

5.2.1. A data structure for disjunctive abstract domains

It turns out that a disjunctive abstract domain $A_{\leqslant} \in \mathrm{dAbs}(\wp(\Sigma))$ can be represented through the partition $\mathrm{par}(A) \in \mathrm{Part}(\Sigma)$ induced by A and the following relation $\trianglelefteq_A$ on $\mathrm{par}(A)$:

$$\forall B_1, B_2 \in \mathrm{par}(A), B_1 \trianglelefteq_A B_2 \text{ iff } \gamma(\alpha(B_1)) \subseteq \gamma(\alpha(B_2)).$$

It is clear that this gives rise to a partial order relation because if $B_1, B_2 \in \mathrm{par}(A)$ and $\gamma(\alpha(B_1)) = \gamma(\alpha(B_2))$ then we can pick up $s_1 \in B_1$ and $s_2 \in B_2$ so that $\gamma(\alpha(\{s_1\})) = \gamma(\alpha(B_1)) = \gamma(\alpha(B_2)) = \gamma(\alpha(\{s_2\}))$, namely s_1 and s_2 are equivalent according to $\mathrm{par}(A)$ and therefore $B_1 = B_2$. The poset $\langle \mathrm{par}(A), \trianglelefteq_A \rangle$ is denoted by $\mathrm{poset}(A)$. It turns out that a disjunctive abstract domain can always be represented by this poset, namely the closure operator induced by A can be defined in terms of $\mathrm{poset}(A)$ as follows.

Lemma 5.2. *Let* $A \in \mathrm{dAbs}(\wp(\Sigma))$. *For any* $S \subseteq \Sigma, \gamma_A(\alpha_A(S)) = \cup\{B \in \mathrm{par}(A) \mid \exists C \in \mathrm{par}(A).\ C \cap S \neq \varnothing,\ B \trianglelefteq_A C\}$.

Proof. ($\subseteq$) Consider any $x \in \gamma_A(\alpha_A(S)) = \cup_{s \in S} \gamma_A(\alpha_A(\{s\}))$. Then, there exists some $s \in S$ such that $x \in \gamma_A(\alpha_A(\{s\}))$. We consider $B_x, B_s \in \mathrm{par}(A)$ such that $x \in B_x$ and $s \in B_s$. Then, $B_s \cap S \neq \varnothing$ and $B_x \trianglelefteq_A B_s$ because $\gamma_A(\alpha_A(B_x)) = \gamma_A(\alpha_A(\{x\})) \subseteq \gamma_A(\alpha_A(\{s\})) = \gamma_A(\alpha_A(B_s))$.
($\supseteq$) Let $B, C \in \mathrm{par}(A)$ such that $s \in C \cap S$ and $B \trianglelefteq_A C$. Then, $B \subseteq \gamma_A(\alpha_A(B)) \subseteq \gamma_A(\alpha_A(C)) = \gamma_A(\alpha_A(\{s\})) \subseteq \gamma_A(\alpha_A(S))$. □

Example 5.3. Some examples of posets that represent disjunctive abstract domains are depicted in Fig. 3.

(1) The disjunctive abstract domain $A_1 = \{\varnothing, [45], [12345]\}$ is such that $\mathrm{par}(A_1) = \{[123], [45]\}$.
(2) The disjunctive domain $A_2 = \{\varnothing, [45], [123], [12345]\}$ induces the same partition $\{[123], [45]\}$, while $\mathrm{poset}(A_2)$ is discrete.
(3) The disjunctive abstract domain $A_3 = \{\varnothing, [4], [5], [45], [12345]\}$ induces the partition $\mathrm{par}(A_3) = \{[123], [4], [5]\}$.
(4) The disjunctive abstract domain $A_4 = \{\varnothing, [45], [145], [245], [1245], [12345]\}$ induces the partition $\mathrm{par}(A_4) = \{[1], [2], [3], [45]\}$.

A disjunctive abstract domain $A \in \mathrm{dAbs}(\wp(\Sigma))$ is thus represented by $\mathrm{poset}(A)$. This means that our implementation of $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ maintains and refines a partition $\mathrm{par}(A)$ and an order relation on $\mathrm{par}(A)$. Let us describe how this can be done.

5.2.2. Implementation

Any state $s \in \Sigma$ is represented by a record `State` that contains a pointer field `block` that points to the block of the current partition $\mathrm{par}(A)$ that includes s and a field `pre` that represents $\mathrm{pre}_{\rightarrow}(\{s\})$ as a list of pointers to the states in $\mathrm{pre}_{\rightarrow}(\{s\})$. The whole state space Σ is represented as a doubly linked list `states` of `State` so that insertion/removal can be done in $O(1)$. The ordering in the list `states` matters and may change during computation.

Any block B of the partition $\mathrm{par}(A) \in \mathrm{Part}(\Sigma)$ is represented by a record `Block` that contains the following fields:

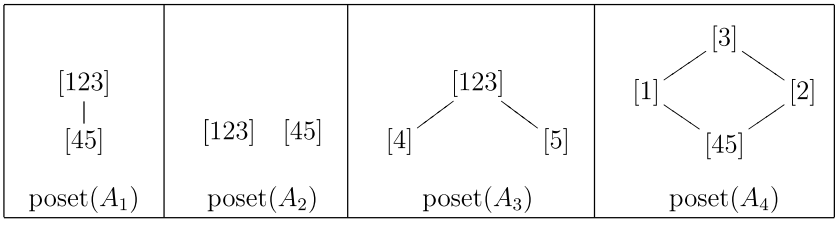

Fig. 3. Disjunctive abstract domains as posets.

- `first` and `last` are pointers to `State` such that the block B consists of all the states in the interval [`first`,`last`] of the list `states`. When a state is either added to or removed from a block, the ordering in the list `states` changes accordingly and this can be done in $O(1)$.
- `less` is a linked list of pointers to `Block`. At the end of any refinement step, the list `less` for some block B contains all the blocks $C \in \mathrm{par}(A)$ which are less than or equal to B, i.e., such that $C \trianglelefteq_A B$. In particular, the list `less` is always nonempty because `less` always includes B itself.
- `intersection` is a pointer to `Block` which is set by the procedure `split` that splits the current partition with respect to a set.
- `changedImage` is a boolean flag which is set by the procedure `orderUpdate`.

The blocks of the current partition $\mathrm{par}(A)$ are represented as a doubly linked list `P` of `Block`.

Let us face the problem of refining a disjunctive abstract domain A to $A' = \mathbb{D}(\gamma(A) \cup \{S\})$ for some $S \subseteq \Sigma$. If $P, P' \in \mathrm{Part}(\Sigma)$, $P' \preceq P$ and $B \in P'$ then let $\mathrm{parent}_P(B) \in P$ (when clear from the context the subscript P is omitted) denote the unique block in P (possibly B itself) that includes B. The following key result provides the basis for designing an algorithm that updates $\mathrm{poset}(A)$ to $\mathrm{poset}(A')$.

Lemma 5.4. *Let* $A \in \mathrm{dAbs}(\wp(\Sigma))$, $S \subseteq \Sigma$ *and* $A' = \mathbb{D}(\gamma(A) \cup \{S\}) \in \mathrm{dAbs}(\wp(\Sigma))$. *Let* $P = \mathrm{par}(A) \in \mathrm{Part}(\Sigma)$ *and* $P' = \mathrm{PTsplit}(S, P) \in \mathrm{Part}(\Sigma)$. *Then*, $\mathrm{poset}(A') = \langle P', \trianglelefteq_{A'} \rangle$, *where for any* $B', C' \in P'$:

(i) *if* $B' \cap S = \varnothing$ *then* $C' \trianglelefteq_{A'} B' \Leftrightarrow C' \subseteq \gamma_A(\alpha_A(\mathrm{parent}(B')))$;
(ii) *if* $B' \cap S \neq \varnothing$ *then* $C' \trianglelefteq_{A'} B' \Leftrightarrow C' \subseteq \gamma_A(\alpha_A(\mathrm{parent}(B'))) \cap S$.

Proof. Let $\mu = \gamma_A \circ \alpha_A$ and $\mu' = \gamma_{A'} \circ \alpha_{A'}$. We first observe that if $x \in S$ then $\mu'(\{x\}) = \mu(\{x\}) \cap S$, while if $x \notin S$ then $\mu'(\{x\}) = \mu(\{x\})$. We then show the following statement: for any $x, y \in \Sigma$,

$$\mu'(\{x\}) \subseteq \mu'(\{y\}) \text{ iff } \mu(\{x\}) \subseteq \mu(\{y\}) \ \&\ (y \in S \Rightarrow x \in S). \tag{$*$}$$

($\Rightarrow$) Since $\mu' \sqsubseteq \mu$, we have that $\mu \circ \mu' = \mu$ so that $\mu(\{x\}) = \mu(\mu'(\{x\})) \subseteq \mu(\mu'(\{y\})) = \mu(\{y\})$. Moreover, if $y \in S$ then $x \in \mu'(\{x\}) \subseteq \mu'(\{y\}) \subseteq \mu'(S) = S$.
($\Leftarrow$) If $y \in S$ then $x \in S$ so that $\mu'(\{x\}) = \mu(\{x\}) \cap S \subseteq \mu(\{y\}) \cap S = \mu'(\{y\})$. If instead $y \notin S$ then $\mu'(\{x\}) \subseteq \mu(\{x\}) \subseteq \mu(\{y\}) = \mu'(\{y\})$.

It is then simple to show that $P' = \mathrm{PTsplit}(S, P) = \mathrm{par}(A')$. In fact, $x \equiv_{A'} y$ iff $\mu'(\{x\}) = \mu'(\{y\})$ and, by $(*)$, this happens iff $\mu(\{x\}) = \mu(\{y\})$ and $x \in S \Leftrightarrow y \in S$, namely iff x and y belong to the same block of $\mathrm{PTsplit}(S, P)$. It is simple to derive from $(*)$ the following statement: for any $B', C' \in P'$,

$$\mu'(C') \subseteq \mu'(B') \text{ iff } \mu(C') \subseteq \mu(B') \ \&\ (B' \cap S \neq \varnothing \Rightarrow C' \cap S \neq \varnothing). \tag{$\ddagger$}$$

Let us now show points (i) and (ii). Let us observe that for any $B' \in P'$, since $P' \preceq P = \mathrm{par}(A)$, we have that $\mu(B') = \mu(\mathrm{parent}(B'))$.
(i) Assume that $B' \cap S = \varnothing$. If $C' \trianglelefteq_{A'} B'$, i.e., $\mu'(C') \subseteq \mu'(B')$, then, by $(\ddagger)$, $\mu(C') \subseteq \mu(B')$ so that $C' \subseteq \mu(C') \subseteq \mu(B') = \mu(\mathrm{parent}(B'))$. On the other hand, if $C' \subseteq \mu(\mathrm{parent}(B')) = \mu(B')$ then $\mu(C') \subseteq \mu(B')$ and $B' \cap S \neq \varnothing \Rightarrow C' \cap S \neq \varnothing$ so that, by $(\ddagger)$, $\mu'(C') \subseteq \mu'(B')$, i.e., $C' \trianglelefteq_{A'} B'$.
(ii) Assume that $B' \cap S \neq \varnothing$. If $C' \trianglelefteq_{A'} B'$, i.e., $\mu'(C') \subseteq \mu'(B')$, then, by $(\ddagger)$, $\mu(C') \subseteq \mu(B')$ and $C' \cap S \neq \varnothing$, namely $C' \subseteq S$. Also, $C' \subseteq \mu(C') \subseteq \mu(B') = \mu(\mathrm{parent}(B'))$ so that $C' \subseteq \mu(\mathrm{parent}(B')) \cap S$. On the other hand, if $C' \subseteq \mu(\mathrm{parent}(B')) \cap S = \mu(B') \cap S$ then $C' \cap S \neq \varnothing$. Also, from $C' \subseteq \mu(B')$ we obtain $\mu(C') \subseteq \mu(B')$. Thus, by $(\ddagger)$, we obtain $\mu'(C') \subseteq \mu'(B')$, i.e., $C' \trianglelefteq_{A'} B'$. □

A refinement step $\mathrm{refine}^{\mathrm{dAbs}}_{\mathrm{pre}}(S, A) = A'$ is thus implemented through the following two main steps:

(A) Update the partition $\mathrm{par}(A)$ to $\mathrm{PTsplit}(S, \mathrm{par}(A))$;
(B) Update the order relation $\trianglelefteq_A$ on $\mathrm{par}(A)$ to $\trianglelefteq_{A'}$ on $\mathrm{PTsplit}(S, \mathrm{par}(A))$ by using Lemma 5.4.

The procedure `split(S)` in Fig. 4 splits the current partition $P \in \mathrm{Part}(\Sigma)$ with respect to a splitter $S \subseteq \Sigma$. Initially, each block $B \in P$ has the field `intersection` set to `NULL`. At the end of `split(S)`, the partition P is updated to its refinement $P' = \mathrm{PTsplit}(S, P)$ where for any $B \in P$:

```
1  /* P is the current partition, S is a list of pointers to State */
2  split(S) {
3    forall state in S do {
4        Block* B = state->block;
5        if (B->intersection==NULL) then {
6           B->intersection = new Block;
7           P.append(B->intersection);
8           B->intersection->intersection = B->intersection;
9           B->intersection->less = copy(B->less);
10          B->intersection->changedImage = false;
11       }
12       move(state,B,B->intersection);
13       if (B = ∅) then { /* case: B ⊆ S */
14          B->first = B->intersection->first; B->last = B->intersection->last;
15          P.remove(B->intersection);
16          delete B->intersection
17          B->intersection = B;
18       }
19   }
20 }
```

```
1  /* P is the current partition after a call to split(S) */
2  orderUpdate() {
3    forall B in P do
4        if (B∩S = ∅) then
5           forall C in B->less
6              if (C ≠ parent(C)) then (B->less).append(parent(C)∩S);
7        else /* case: B∩S ≠ ∅, i.e. B ⊆ S */
8           forall C in B->less {
9             if (C⊆S) then continue;
10            /* case: C∩S = ∅ */
11            (B->less).remove(C);
12            if (parent(C)∩S ≠ ∅) (B->less).append(parent(C)∩S);
13            B->changedImage = true;
14          }
15 }
```

Fig. 4. The procedures `split(S)` and `orderUpdate()`.

– If $\varnothing \subsetneq B \cap S \subsetneq B$ then B is modified to $B \setminus S$ by repeating the move statement at line 12 and the newly allocated block $B \cap S$ at line 6 is appended at line 7 to the end of the current list of blocks;
– If $B \cap S = B$ or $B \cap S = \varnothing$ then B is not modified.

Moreover, the field `intersection` of any $B' \in P' = \mathrm{PTsplit}(S, P)$ is set as follows:

(1) If $B' \in P \cap P'$ and $B' \cap S = \varnothing$ then B'`->intersection` = `NULL` because `split(`S`)` does not modify the record B'.
(2) If $B' \in P \cap P'$ and $B' \cap S \neq \varnothing$ (i.e., $B' \subseteq S$) then B'`->intersection` $= B'$ (line 17).
(3) If $B' \in P' \setminus P$ and $B' \cap S = \varnothing$ (i.e., $B' = \mathrm{parent}(B') \setminus S$) then B'`->intersection` $= \mathrm{parent}(B') \cap S$ (line 6).
(4) If $B' \in P' \setminus P$ and $B' \cap S \neq \varnothing$ (i.e., $B' = \mathrm{parent}(B') \cap S$) then B'`->intersection` $= B'$ (line 8).

Note that for the "old" blocks in P, `split(`S`)` does not modify the corresponding list of pointers `less`, while the list `less` for a newly allocated block $B \cap S$ is a copy of the list `less` of B (line 9). Also observe that blocks that are referenced by pointers in some `less` field may well be modified.

```
1   /* The list Atoms represents the set {⟦p⟧_K ⊆ Σ | p ∈ AP} */
2   /* P is initialized to the single block partition */
3   Partition P = (Σ); Σ->less = {Σ};
4
5   forall S in Atoms do {
6      split(S); orderUpdate();
7      split(CS); orderUpdate();
8   }
9   forall B in P do {
10     State* X = image(B);
11     State* S = NULL;
12     forall s in X do S.append(s->pre);
13     split(S);
14     orderUpdate();
15     forall B in P do {
16        B->intersection = NULL;
17        if (B->changedImage) {B->changedImage = false; P.moveAtTheEnd(B);}
18     }
19  }
```

Fig. 5. Implementation of $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$.

The procedure `orderUpdate()` in Fig. 4 is called after `split(S)` to update the `less` fields in order to represent the refined poset $\langle P', \trianglelefteq_{A'}\rangle$ defined in Lemma 5.4. By exploiting the above points (1)–(4), let us observe the following points about the procedure `orderUpdate()` whose current partition represents $P' = \mathrm{PTsplit}(S, P)$.

(5) For all blocks $B' \in P'$, the test $B' \cap S = \varnothing$ at line 4 is coded as B'`->intersection` $\neq B'$.
(6) The test $C \neq \mathrm{parent}(C)$ at line 6 is coded as C`->intersection` $\neq$ `NULL` and C`->intersection` $\neq C$.
(7) The block $\mathrm{parent}(C) \cap S$ at lines 6 and 10 is C`->intersection`.
(8) The test $C \subseteq S$ at line 9 is equivalent to $C \cap S \neq \varnothing$ and is thus coded as C`->intersection` $= C$.
(9) Lines 4–6 implement the case (i) of Lemma 5.4.
(10) Lines 7–14 implement the case (ii) of Lemma 5.4.

Moreover, if for some blocks $B, C \in P'$ we have that $B \subseteq S$ and C belongs to the list B`->less` and $C \cap S = \varnothing$—namely, we are in the case of line 10—then, by Lemma 5.4, $\gamma_{A'}(\alpha_{A'}(B)) \subsetneq \gamma_A(\alpha_A(B))$, that is the image of B changed. For these blocks B, the flag B`->changedImage` is set to `true`.

Finally, let us notice that the sequence of disjunctive abstract domains computed by some run of $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ is decreasing, namely if A and A' are, respectively, the current and next disjunctive abstract domains then $A' \sqsubseteq A$. As a consequence, if an image $\gamma_A(\alpha_A(B))$, for some $B \in \mathrm{par}(A)$, is not a refiner for A and B remains a block in the next refined partition $\mathrm{par}(A')$ then $\gamma_{A'}(\alpha_{A'}(B))$ cannot be a refiner for A'. Thus, a correct strategy for finding refiners consists in scanning the list of blocks of the current partition P while in any refinement step from A to A', after calling `split(S)`, all the blocks $B \in \mathrm{par}(A')$ whose image changed are moved to the tail of P. This leads to the implementation of $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$ described in Fig. 5.

Theorem 5.5. *The algorithm in Fig. 5 computes simulation equivalence P_{sim} on $\mathcal{K}$ in $O(|P_{\mathrm{sim}}|^2 \cdot (|P_{\mathrm{sim}}|^2 + |{\rightarrow}|))$-time and $O(|\Sigma| \log(|P_{\mathrm{sim}}|) + |P_{\mathrm{sim}}|^2)$-space.*

Proof. We have shown above that the algorithm in Fig. 5 is a correct implementation of $\mathrm{GPT}^{\mathrm{dAbs}}_{\mathrm{pre}}$. Let us observe the following points.

(1) For any block $B \in P$, by Lemma 5.2, `image(B)` at line 10 can be computed in the worst case by scanning each edge of the order relation $\trianglelefteq_A$ on $P = \mathrm{par}(A)$, namely in $O(|P|^2)$ time. Since any current partition is coarser than P_{sim}, it turns out that `image(B)` can be computed in $O(|P_{\mathrm{sim}}|^2)$-time.
(2) The list of pointers S at lines 11–12 representing $\mathrm{pre}_{\rightarrow}(\gamma_A(B))$ can be computed in the worst case by traversing the whole transition relation, namely in $O(|{\rightarrow}|)$-time.

(3) For any $S \subseteq \Sigma$, `split(S)` at line 13 is computed in $O(|S|)$-time.
(4) `orderUpdate()` at line 14 is computed in the worst case by scanning each edge of the order relation $\trianglelefteq_A$ on $P = \mathrm{par}(A)$, namely in $O(|P|^2)$ time, and therefore in $O(|P_{\mathrm{sim}}|^2)$-time.
(5) The for loop at line 15 is computed in $O(|P|)$-time and therefore in $O(|P_{\mathrm{sim}}|)$-time.

Thus, an iteration of the for-loop takes $O(2|P_{\mathrm{sim}}|^2 + |{\rightarrow}| + |S| + |P_{\mathrm{sim}}|)$-time, namely, since $|S| \leqslant |{\rightarrow}|$, $O(|P_{\mathrm{sim}}|^2 + |{\rightarrow}|)$-time.
In order to prove that the time complexity is $O(|P_{\mathrm{sim}}|^2 \cdot (|P_{\mathrm{sim}}|^2 + |{\rightarrow}|))$, let us show that the number of iterations of the for-loop is in $O(|P_{\mathrm{sim}}|^2)$. Let $\{A_i\}_{i\in[1,k]} \in \mathrm{dAbs}(\wp(\Sigma))$ be the sequence of different disjunctive abstract domains computed in some run of the algorithm and let $\{\mu_i\}_{i\in[1,k]} \subseteq \mathrm{uco}(\wp(\Sigma))$ be the corresponding sequence of disjunctive uco's. Thus, for any $i \in [1,k)$, $\mu_{i+1} \sqsubset \mu_i$ and $P_{\mathrm{sim}} = \mathrm{par}(\mu_k)$. Hence, for any $i \in [1,k]$, $P_{\mathrm{sim}} \preceq \mathrm{par}(\mu_i)$, so that for any $B \in P_{\mathrm{sim}}$, $\mu_i(B) = \cup_{j\in J} B_j$ for some set of blocks $\{B_j\}_{j\in J} \subseteq P_{\mathrm{sim}}$. We know that for any $i \in [1,k)$ there exists some block $B \in \mathrm{par}(\mu_i)$ whose image changes, namely $\mu_{i+1}(B) \subsetneq \mu_i(B)$. Note that $\mu_{i+1}(B) \subsetneq \mu_i(B)$ holds for some $B \in \mathrm{par}(\mu_i)$ if and only if $\mu_{i+1}(B) \subsetneq \mu_i(B)$ holds for some $B \in P_{\mathrm{sim}}$. Clearly, for any block $B \in P_{\mathrm{sim}}$, this latter fact can happen at most $|P_{\mathrm{sim}}|$ times. Consequently, the overall number of blocks that in some iteration of the for-loop change image is bounded by $\sum_{B\in P_{\mathrm{sim}}} |P_{\mathrm{sim}}| = |P_{\mathrm{sim}}|^2$. Hence, the overall number of blocks that are scanned by the for-loop is bounded by $|\mathrm{par}(\mu_1)| + |P_{\mathrm{sim}}|^2$ and therefore the overall number of iterations of the for-loop is in $O(|P_{\mathrm{sim}}|^2)$.
The input of the algorithm is the Kripke structure $\mathcal{K}$, that is the list `states` and for each state the list `pre` of its predecessors. In each iteration of the for loop we keep in memory all the fields of the records `State`, that need $O(|\Sigma| \log(|P_{\mathrm{sim}}|))$-space, the current partition, that needs $O(|P_{\mathrm{sim}}|)$-space, and the order relation $\trianglelefteq_A$, that needs $O(|P_{\mathrm{sim}}|^2)$-space. Thus, the overall space complexity is $O(|\Sigma| \log(|P_{\mathrm{sim}}|) + |P_{\mathrm{sim}}|^2)$. □

5.3. A language expressing reachability

Let us consider the following language $\mathcal{L}$ which is generated by the existential "finally" operator EF together with propositional logic:

$$\varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EF}\varphi.$$

Given a Kripke structure $\mathcal{K} = (\Sigma, {\rightarrow}, \ell)$, the interpretation $\mathbf{EF} : \wp(\Sigma) \to \wp(\Sigma)$ of the reachability operator EF is as usual: $\mathbf{EF}(S) \stackrel{\mathrm{def}}{=} \mathbf{EU}(\Sigma, S)$. It turns out that the coarsest strongly preserving partition $P_{\mathcal{L}}$ coincides with Milner's weak bisimulation equivalence on unlabeled transition systems. Weak bisimulation [25] is a weakening of bisimulation on labeled transition systems which allows any finite number of invisible τ-labeled actions before or after a simulation step. On unlabeled Kripke structures, weak bisimulation simply allows to simulate one transition step through any finite number of transition steps. Hence, a relation $R \subseteq \Sigma \times \Sigma$ is a weak bisimulation on a Kripke structure $\mathcal{K} = (\Sigma, {\rightarrow}, \ell)$ when R is a bisimulation on the Kripke structure $\mathcal{K}^* = (\Sigma, {\rightarrow^*}, \ell)$ where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$. Analogously to bisimulation, the largest weak bisimulation relation exists and is an equivalence relation called weak bisimulation equivalence, whose corresponding partition is denoted by $P_{\mathrm{wbis}} \in \mathrm{Part}(\Sigma)$. Then, it turns out that $P_{\mathrm{wbis}} = P_{\mathcal{L}}$. It would not be too difficult to demonstrate this latter equivalence, for example by adapting the proof given in [28, Corollary 7.4] for stuttering equivalence.

To the best of our knowledge, no specific algorithm for computing weak bisimulation equivalence on unlabeled Kripke structures exists. One naïve algorithm simply consists in first computing the reflexive-transitive closure of the transition relation and successively computing bisimulation equivalence through PT. However, this approach would require to store the reflexive-transitive closure of the transition relation and in general this would be a space bottleneck. We provide here a specific algorithm for computing $P_{\mathcal{L}}$ which is obtained as an instantiation of GPT. Since $\mathcal{L}$ includes propositional logic, by Corollary 4.8, it turns out that the instance $\mathrm{GPT}^{\mathrm{Part}}_{\mathbf{EF}}$ allows to compute the coarsest strongly preserving partition $P_{\mathcal{L}}$, namely $\mathrm{GPT}^{\mathrm{Part}}_{\mathbf{EF}}(P_\ell) = P_{\mathcal{L}}$.

It turns out that block refiners are enough, namely

$$\mathrm{BlockRefiners}^{\mathrm{Part}}_{\mathbf{EF}}(P) = \{B \in P \mid P \not\curlywedge \{\mathbf{EF}(B), \complement(\mathbf{EF}(B))\} \prec P\}.$$

In fact, note that $\mathrm{BlockRefiners}^{\mathrm{Part}}_{\mathbf{EF}}(P) = \varnothing$ iff $\mathrm{Refiners}^{\mathrm{Part}}_{\mathbf{EF}}(P) = \varnothing$, so that, by exploiting Corollary 4.7, we have that $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}(P_\ell) = P_{\mathcal{L}}$. The optimized algorithm $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ is as follows.

ALGORITHM: $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$

Input: partition $P \in \mathrm{Part}(\Sigma)$
while $(\mathrm{BlockRefiners}^{\mathrm{Part}}_{\mathbf{EF}}(P) \neq \varnothing)$ **do**
  choose $B \in \mathrm{BlockRefiners}^{\mathrm{Part}}_{\mathbf{EF}}(P)$;
  $P := P \curlywedge \{\mathbf{EF}(B), \complement(\mathbf{EF}(B))\}$;
Output: P

5.3.1. Implementation

The key point in implementing $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ is the following property of "stability under refinement": for any $P, Q \in \mathrm{Part}(\Sigma)$,

$$\text{if } Q \preceq P \text{ and } B \in P \cap Q \text{ then } P \curlywedge \{\mathbf{EF}(B), \complement(\mathbf{EF}(B))\} = P \text{ implies } Q \curlywedge \{\mathbf{EF}(B), \complement(\mathbf{EF}(B))\} = Q.$$

As a consequence of this property, if some block B of the current partition P_{curr} is not a **EF**-refiner for P_{curr} and B remains a block of the next partition P_{next} then B cannot be a **EF**-refiner for P_{next}.

This suggests an implementation of $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ based on the following points:

(1) The current partition P is represented as a doubly linked list of blocks (so that a block removal can be done in $O(1)$-time).
(2) This list of blocks P is scanned from the beginning in order to find block refiners.
(3) When a block B of the current partition P is split into two new blocks B_1 and B_2 then B is removed from the list P and B_1 and B_2 are appended to the end of P.

These ideas lead to the implementation $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ described in Fig. 6. As a preprocessing step we compute the DAG of the strongly connected components (s.c.c.'s) of the directed graph $(\Sigma, \rightarrow)$, denoted by $(P_{\mathrm{scc}}, \rightarrow_{\mathrm{scc}})$. This is done by the depth-first Tarjan's algorithm [31] in $O(|\rightarrow|)$-time. This preprocessing step is done because if $x \in \mathbf{EF}(S)$, for some $x \in \Sigma$ and $S \subseteq \Sigma$, then the whole block B_x in the partition P_{scc} that contains x—i.e., the strongly connected component containing x—is contained in $\mathbf{EF}(S)$; moreover, let us also observe that $\mathbf{EF}(\{x\}) = \mathbf{EF}(B_x)$. The algorithm then proceeds by scanning the list of blocks P and by performing the following three steps: (1) for the current block B of the current partition P, we first compute the set B_{scc} of s.c.c.'s that contain some state in B; (2) we then compute $\mathbf{EF}(B_{\mathrm{scc}})$ in the DAG $(P_{\mathrm{scc}}, \rightarrow_{\mathrm{scc}})$ because $\mathbf{EF}(B) = \bigcup \mathbf{EF}(B_{\mathrm{scc}})$; (3) finally, we split the current partition P with respect to the splitter $\mathbf{EF}(B)$. The computation of $\mathbf{EF}(B_{\mathrm{scc}})$ is performed by the simple procedure **computeEF**(B_{scc}) in Fig. 6 in $O(|\rightarrow_{\mathrm{scc}}|)$-time while splitting P with respect to S is done by the procedure **split**(S, P) in Fig. 6 in $O(|S|)$-time. It turns out that this implementation runs in $O(|\rightarrow||\Sigma|)$-time.

Theorem 5.6. *The implementation of* $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ *in Fig. 6 is correct and runs in* $O(|\rightarrow||\Sigma|)$*-time.*

Proof. Let us show the following points.

(1) Each iteration of the scan loop takes $O(|\rightarrow|)$-time.
(2) The number of iterations of the scan loop is in $O(|\Sigma|)$.

(1) Let B be the current block while scanning the current partition P. The set $B_{\mathrm{scc}} = \{C \in P_{\mathrm{scc}} \mid B \cap C \neq \varnothing\}$ is determined in $O(|B|)$-time simply by scanning the states in B. The computation of $\mathbf{EF}(B_{\mathrm{scc}})$ in the DAG of s.c.c.'s $(P_{\mathrm{scc}}, \rightarrow_{\mathrm{scc}})$ takes $O(|\rightarrow_{\mathrm{scc}}|)$-time, the union $S = \bigcup \mathbf{EF}(B_{\mathrm{scc}})$ takes $O(|S|)$-time, while splitting P with respect to S takes $O(|S|)$-time. Thus, each iteration is done in $O(|B| + |\rightarrow_{\mathrm{scc}}| + 2|S|) = O(|\rightarrow| + |\Sigma|) = O(|\rightarrow|)$, since $|\Sigma| \leqslant |\rightarrow|$.

```
ALGORITHM: IGPT^Part_EF
Input: Transition System (Σ, →), List⟨Blocks⟩ P
(P_scc, →_scc) := scc(Σ, →);
scan B in P do
    List⟨BlocksOfBlocks⟩ B_scc := {C ∈ P_scc | B ∩ C ≠ ∅};
    List⟨States⟩ S := ⋃ computeEF(B_scc);
    S := ⋃ computeEF(B_scc);
    split(S, P);
Output: P

List⟨States⟩ computeEF(List⟨States⟩ S) {
    List⟨States⟩ result;
    scan s in S do {result.append(s); mark(s); }
    scan s in result do
        forall r ∈ pre({s}) do
            if (r isNotMarked) then {result.append(r); mark(r);}
    return result;
}

split(List⟨States⟩ S, List⟨Blocks⟩ P) {
    scan s in S do
        Block B := s.block;
        if (B.intersection = false) then
            B.intersection := true; B.split := true;
            Block B ∩ S := new Block;
            P.append(B ∩ S);
        moveFromTo(s, B, B ∩ S);
        if (B = ∅) then
            B.split := false;
            B := B ∩ S;
            P.remove(B ∩ S);
    scan B in P do
        if (B.split = true) then P.moveAtTheEnd(B);
}
```

Fig. 6. Implementation of $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$.

(2) Let B be the current block of the current partition P_{curr}. Then, the next partition $P_{\mathrm{next}} \preceq P_{\mathrm{curr}}$ is obtained by splitting through $\mathbf{EF}(B)$ a number $k \geqslant 0$ of blocks of P_{curr} so that $|P_{\mathrm{next}}| = |P_{\mathrm{curr}}| + k$, where we also consider the case that $\mathbf{EF}(B)$ is not a splitter for P, namely the case $k = 0$. Recall that any partition P has a certain height $\hbar(P) = |\Sigma| - |P|$ in the lattice $\mathrm{Part}(\Sigma)$ which is bounded by $|\Sigma| - 1$. Thus, after splitting k blocks we have that $\hbar(P_{\mathrm{next}}) = \hbar(P_{\mathrm{curr}}) - k$. The overall number of blocks that are split in some run of the algorithm is therefore bounded by $|\Sigma|$. As a consequence, if $\{P_i\}_{i=0}^{m}$ is the sequence of partitions computed in some run of the algorithm and $\{k_i\}_{i=0}^{m-1}$ is the corresponding sequence of the number of splitting steps for each P_i, where $k_i \geqslant 0$, then $\sum_{i=0}^{m-1} k_i \leqslant |\Sigma|$. Also, at each iteration i the number of new blocks is $2k_i$, so that the overall number of new blocks in some run of the algorithm is $\sum_{i=0}^{m-1} 2k_i \leqslant 2|\Sigma|$. Summing up, the total number of blocks that are scanned by the scan loop is $|P_0| + \sum_{i=0}^{m-1} 2k_i \leqslant |P_0| + 2|\Sigma| \leqslant 3|\Sigma|$ and therefore the number of iterations is in $O(|\Sigma|)$.

Since the computation of the DAG of s.c.c.'s that precedes the scan loop takes $O(|{\rightarrow}|)$-time, the overall time complexity of the algorithm is $O(|{\rightarrow}||\Sigma|)$. □

Model	States	Transitions	Initial Blocks	Final Blocks	BisimEq Blocks	Time
cwi_1_2	4339	4774	27	27	2959	0.04s
cwi_3_14	18548	29104	3	123	123	0.54s
vasy_0_1	1513	2448	3	12	152	0.01s
vasy_10_56	67005	112312	13	18	67005	0.49s
vasy_1_4	5647	8928	7	51	3372	0.10s
vasy_18_73	91789	146086	18	161	70209	3.65s
vasy_25_25	50433	50433	25217	50433	50433	365.01s
vasy_40_60	100013	120014	4	4	100013	0.43s
vasy_5_9	15162	19352	32	2528	13269	2.33s
vasy_8_24	33290	48822	12	6295	30991	18.80s
vasy_8_38	47345	76848	82	13246	47345	5.11s

Fig. 7. Results of the experimental evaluation.

5.3.2. Experimental evaluation

A prototype of the above partition refinement algorithm $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ has been developed in C++. We considered the standard VLTS (Very Large Transition Systems) benchmark suite for our experimental evaluation [32]. The VLTS suite consists of transition systems encoded in the BCG (Binary-Coded Graphs) format where labels are attached to arcs. Since our algorithm needs as input a Kripke structure, namely a transition system where labels are attached to states, we exploited a procedure designed by Dovier et al. [9] that transforms an edge-labeled graph G into a node-labeled graph G' in a way such that bisimulation equivalences on G and G' coincide. This conversion acts as follows: any transition $s_1 \xrightarrow{l} s_2$ is replaced by two transitions $s_1 \to n$ and $n \to s_2$, where n is a new node that is labeled with l. Hence, this transformation grows the size of the graph: the number of transitions is doubled and the number of nodes grows proportionally to the average branching factor of G.

Our experimental evaluation of $\mathrm{IGPT}^{\mathrm{Part}}_{\mathbf{EF}}$ was carried out on an Intel Core 2 Duo 1.86 GHz PC, with 2 GB RAM, running Linux 2.6.20 and GNU g++ 4.1.2. The results are summarised in Fig. 7, where we list, respectively, the name of the original transition system in the VLTS suite, the number of states and transitions of the transformed transition system, the number of blocks of the initial partition, the number of blocks of the final refined partition, the number of bisimulation equivalence classes and the execution time in seconds. In each experiment, the memory used never exceeded 32 MB. The goal of our experiments was simply to assess whether the algorithm can be practically used for Kripke structures of medium size (the sizes here are comparable with those of the experiments reported in [11]). The experiments show that in these cases $P_{\mathcal{L}}$ can be practically computed with a small time cost.

6. Related work

Dams [7, Chapter 5] presents a generic splitting algorithm that, for a given language $\mathcal{L} \subseteq$ ACTL, computes an abstract model $A \in \mathrm{Abs}(\wp(\Sigma))$ that strongly preserves $\mathcal{L}$. This technique is inherently different from ours, in particular because it is guided by a splitting operation of an abstract state that depends on a given formula of ACTL. Additionally, Dams' methodology does not guarantee optimality of the resulting strongly preserving abstract model, as instead we do, because his algorithm may provide strongly preserving models that are too concrete. Dams [7, Chapter 6] also presents a generic partition refinement algorithm that computes a given (behavioural) state equivalence and generalizes PT (i.e., bisimulation equivalence) and Groote and Vaandrager

(i.e., stuttering equivalence) algorithms. This algorithm is parameterized on a notion of splitter corresponding to some state equivalence, while our algorithm is directly parameterized on a given language: the example language given in [7] (a "flat" version of CTL-X) seems to indicate that finding the right definition of splitter for a given language may be a hard task. Gentilini et al. [11] provide an algorithm that solves a so-called generalized coarsest partition problem, meaning that they generalized PT stability to partitions endowed with an acyclic relation (so-called partition pairs). They show that this technique can be instantiated to obtain a logarithmic algorithm for PT stability and an efficient algorithm for simulation equivalence. This approach is very different from ours since the partition refinement algorithm is not driven by strong preservation with respect to some language. Finally, it is also worth citing that Habib et al. [18] show that the technique of iteratively refining a partition by splitting blocks with respect to some pivot set, as it is done in PT, may be generally applied for solving problems in various contexts, ranging from strings to graphs. In fact, they show that a generic skeleton of partition refinement algorithm, based on a partition splitting step with respect to a generic pivot, can be instantiated in a number of relevant cases where the context allows an appropriate choice for the set of pivots.

7. Conclusion and future work

In model checking, the well-known Paige–Tarjan algorithm is used for minimally refining a given state partition in order to obtain a standard abstract model that strongly preserves the branching-time language CTL on some Kripke structure. We designed a generalized Paige–Tarjan algorithm, called GPT, that minimally refines generic abstract interpretation-based models in order to obtain strong preservation for a generic inductive language. Abstract interpretation has been the key tool for accomplishing this task. GPT may be systematically instantiated to classes of abstract models and inductive languages that satisfy some conditions. We showed that some existing partition refinement algorithms can be viewed as an instance of GPT and that GPT may yield new efficient algorithms for computing strongly preserving abstract models, like simulation equivalence.

GPT is parameteric on a domain of abstract models which is an abstraction of the lattice of abstract domains $\mathrm{Abs}(\wp(\Sigma))$. GPT has been instantiated to the lattice $\mathrm{Part}(\Sigma)$ of partitions and to the lattice $\mathrm{dAbs}(\wp(\Sigma))$ of disjunctive abstract domains. It is definitely interesting to investigate whether the GPT scheme can be applied to new domains of abstract models. In particular, models that are abstractions of $\mathrm{Part}(\Sigma)$ could be useful for computing approximations of strongly preserving partitions. As an example, if one is interested in reducing only a portion $S \subseteq \Sigma$ of the state space Σ then we may consider the domain $\mathrm{Part}(S)$ of partitions of S as an abstraction of $\mathrm{Part}(\Sigma)$ in order to get strong preservation only on the portion S.

Acknowledgments

This work was partially supported by the project "Formal methods for specifying and verifying behavioural properties of software systems" funded by University of Padova and by the FIRB project "Abstract interpretation and model checking for the verification of embedded systems".

References

[1] B. Bloom, R. Paige, Transformational design and implementation of a new efficient solution to the ready simulation problem, Sci. Comp. Program. 24 (3) (1995) 189–220.
[2] M.C. Browne, E.M. Clarke, O. Grumberg, Characterizing finite Kripke structures in propositional temporal logic, Theor. Comp. Sci. 59 (1988) 115–131.
[3] D. Bustan, O. Grumberg, Simulation-based minimization, ACM Trans. Comput. Logic 4 (2) (2003) 181–204.
[4] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 1999.
[5] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM POPL, ACM Press, 1977, pp. 238–252.
[6] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the 6th ACM POPL, ACM Press, 1979, pp. 269– 282.

[7] D. Dams, Abstract Interpretation and Partition Refinement for Model Checking, PhD Thesis, Eindhoven University, 1996.
[8] R. De Nicola, F. Vaandrager, Three logics for branching bisimulation, J. ACM 42 (2) (1995) 458–487.
[9] A. Dovier, C. Piazza, A. Policriti, An efficient algorithm for computing bisimulation equivalence, Theor. Comput. Sci. 325 (1) (2004) 45–67.
[10] G. Filé, R. Giacobazzi, F. Ranzato, A unifying view of abstract domain design, ACM Comput. Surv. 28 (2) (1996) 333–336.
[11] R. Gentilini, C. Piazza, A. Policriti, From bisimulation to simulation: coarsest partition problems, J. Autom. Reason. 31 (1) (2003) 73–103.
[12] R. Giacobazzi, E. Quintarelli, Incompleteness, counterexamples and refinements in abstract model checking, in: Proceedings of the 8th SAS, LNCS vol. 2126, 2001, Springer, pp. 356–373.
[13] R. Giacobazzi, F. Ranzato, Refining and compressing abstract domains, in: Proceedings of the 24th ICALP, LNCS, vol. 1256, 1997, Springer, pp. 771–781.
[14] R. Giacobazzi, F. Ranzato, F. Scozzari, Making abstract interpretations complete, J. ACM 47 (2) (2000) 361–416.
[15] R.J. van Glabbeek, The linear time-branching time spectrum I: the semantics of concrete sequential processes, in: Handbook of Process Algebra, Elsevier, 2001, pp. 3–99.
[16] R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics, in: G.X. Ritter (Ed.), Information Processing '89, Elsevier, 1989, pp. 613–618.
[17] J.F. Groote, F. Vaandrager, An efficient algorithm for branching bisimulation and stuttering equivalence, in: Proceedings of the 17th ICALP, LNCS, vol. 443, Springer, 1990, pp. 626–638.
[18] M. Habib, C. Paul, L. Vienot, Partition refinement techniques: an interesting algorithmic tool kit, Int. J. Found. Comput. Sci. 10 (2) (1999) 147–170.
[19] M. Hennessy, R. Milner, Algebraic laws for nondeterminism and concurrency, J. ACM 32 (1) (1985) 137–161.
[20] M.R. Henzinger, T.A. Henzinger, P.W. Kopke, Computing simulations on finite and infinite graphs, in: Proceedings of the 36th FOCS, IEEE Press, 1995, pp. 453–462.
[21] T.A. Henzinger, R. Maujumdar, J.-F. Raskin, A classification of symbolic transition systems, ACM Trans. Comput. Logic 6 (1) (2005) 1–31.
[22] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, 1971, pp. 189–196.
[23] A. Kucera, R. Mayr, Why is simulation harder than bisimulation? in: Proceedings of the 13th CONCUR, LNCS, vol. 2421, Springer, 2002, pp. 594–610.
[24] L. Lamport, What good is temporal logic?, in: Information Processing '83, IFIP North-Holland, 1983, pp. 657–668.
[25] R. Milner, A Calculus of Communicating Systems, LNCS, vol. 92, Springer, 1980
[26] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (6) (1987) 973–989.
[27] F. Ranzato, F. Tapparo, in: An abstract interpretation-based refinement algorithm for strong preservation, in: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), LNCS, vol. 3440, Springer, 2005, pp. 140–156.
[28] F. Ranzato, F. Tapparo, Generalized strong preservation by abstract interpretation, J. Logic Comput. 17 (1) (2007) 157–197.
[29] F. Ranzato, F. Tapparo, A new efficient simulation equivalence algorithm, in: Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS'07), IEEE Press, 2007, pp. 171–180.
[30] L. Tan, R. Cleaveland, Simulation revisited, in: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), LNCS, vol. 2031, 2001, Springer, pp. 480–495.
[31] R.E. Tarjan, Depth-first search and linear graph algorithms, in: SIAM J. Comput. 1 (2) (1972) 146–160.
[32] The VLTS Benchmark Suite. Collection of Very Large Transition Systems, Available from: http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html.